Modeling and Analysis of Real Time Systems with Preemption, Uncertainty and Dependency

Marcelo Zanconi

Université Joseph Fourier

N° attribué par la bibliothèque
|__ /__ /__ /__ /__ /__ /__ /__ /__ /__ /

T H È S E

pour obtenir le grade de

DOCTEUR DE L'UJF

Spécialité : "INFORMATIQUE : SYSTÈMES ET COMMUNICATION"

préparée au laboratoire VERIMAG
dans le cadre de l'**École doctorale "MATHÉMATIQUES, SCIENCES ET TECHNOLOGIES DE L'INFORMATION, INFORMATIQUE"**

présentée et soutenue publiquement

par

Marcelo ZANCONI

le 22 Juin 2004

Titre :

Modeling and Analysis of Real Time Systems with Preemption, Uncertainty and Dependency

(Modélisation et Analyse de Systèmes Temps Réel, avec Preemption, Incertitude et Dependences)

Directeur de thèse :

Sergio YOVINE

JURY

Dominique Duval	Presidente
Alfredo Olivero	Rapporteur
Ahmed Bouajjani	Rapporteur
Philippe Clauss	Examinateur
Jacques Pulou	Examinateur

Contents

List of Figures

Remerciements

Finalement, le jour est arrivé ou on décide d'écrire quelques mots de remerciements; ça veut dire que la thèse est finie (plus ou moins...), qu'on a fait un tas de copies provisoires, en espérant que chaque copie soit "la dernière", que les rapports sont arrivés, qu'on attaque la soutenance et que ça a rien de "provisoire". On se dit alors... pourquoi pas penser aux remerciements? J'y vais!

Un tas de noms viennent dans ma tête. Je m'organise:

Je voudrais remercier mon directeur, Sergio Yovine, qui m'a soutenu pendant ces 40 mois; il a été toujours là pour m'aider, me conseiller, me guider, m'enseigner, mais surtout pour me donner confiance en ce que je faisais et pour me transmettre que dans la recherche on pousse toujours les limites, les frontières de l'inconnu. Merci, encore!

Je voudrais remercier tout le personnel de Verimag: chercheurs, enseignants, ingénieurs, étudiants et personnel administratif. Avec chaque une et chaque un, je sens que j'ai partagé un moment: un café, un séminaire, une discussion technique, un peu de philosophie, quelques opinions politiques ou même des échanges culinaires! C'est sûr qu'après presque trois ans et demi de "vie en commun" vous allez me manquer... Un Grand Merci à Joseph Sifakis pour m'avoir accueilli chaleureusement et au personnel administratif qui m'a tant aidé avec mon français!

Un grand merci, à la Région Rhônes Alpes qui m'a généreursement soutenu financièrement pendant trois ans et au Gouverment Français qui m'a facilité énormement mon déplacement en France et toutes les démarches administratives de cartes de séjour et visas.

Un énorme merci à Pierre qui est mon soutien; son amour, sa bonne humeur toujours euphorique et positive et son bon êtat d'esprit m'a enormement aidé à faire face à la distance entre la France et l'Argentine.

Merci à ma famille en Argentine; même si la décision a été très dure, ils ont compris que la réalisation d'une thèse et l'expérience de vivre à l'étranger vaut le malheur qui provoque la distance.

Un enorme merci à tous mes amis d'Argentine qui jour après jour sont là "derrière" l'ecran de mon ordinateur avec un e-mail, un mot d'encouragement, une blague.

Et bien évidemment, merci aussi à mes amis de Grenoble avec qui je partage un week-end, des bières et toutes les belles choses de cette ville magnifique.

Agradecimientos

Esta es una parte importante de mi tesis, aquella en la que agradezco a las personas que me ayudaron y me acompañaron en esta tarea y por ello esta escrita en mi lengua materna.

Agradezco en primer lugar a mi director de tesis, Sergio Yovine quien me respaldo enormemente durante estos 40 meses de labor; estuvo siempre alli, para ayudarme, aconsejarme, guiarme y sobre todo para darme confianza en lo que haciamos y transmitirme que en el campo de la investigacion, como en muchos otros, hay que saber cortar barreras y franquear los limites de lo desconocido. Muchas gracias!

Quiero agradecer igualmente a todo el personal de Verimag: investigadores, profesores, ingenieros, estudiantes y personal administrativo. Con cada uno siento que comparti un momento agradable: los mediodias de crucigramas, los seminarios, la lectura comentada del diario y hasta intercambios culinarios! Gracias especialmente a Joseph Sifakis quien me recibio calurosamente en su laboratorio y a todo el personal administrativo que tanto me ha ayudado con el frances!!

Mi agradecimiento va tambien a la Region Rhônes-Alpes y al Gobierno Frances por su ayuda financiera durante todos estos años y por facilitarme enormemente los tramites administrativos de estadia.

Un profundo agradecimiento para Pierre por su respaldo y apoyo constante; su amor, su buen humor siempre euforico y entusiasta y su buen estado de espiritu han facilitado enormemente el afrontar la distancia entre Argentina y Francia.

Mil y mil gracias a mi familia en Argentina; aun cuando la decision de trasladarse al extranjero fue dificil de aceptar, pronto comprendieron que la importancia de realizar la tesis, bien vale la pena la desazón.

Gracias a la Universidad Nacional del Sur y en especial al Departamento de Ciencias e Ingenieria de la Computacion por su apoyo incondicional a mi decision de realizar una tesis en el extranjero y a todos mis profesores que me apoyaron.

Enorme y profundo agradecimento va tambien para mis amigos en Argentina; estuvieron (y estan) siempre alli, "detras" de la pantalla, con un mensaje, una palabra de aliento, un chiste para los momentos de flojedad.

Y por supuesto, muchas gracias tambien a mis amigos de Grenoble con quienes comparto los fines de semana, innumerables cervezas y todas las lindas cosas del *savoir vivre* frances.

Chapter 1

Introducing the actors

Résumé

Les systèmes temps-réel, STR, sont soumis à des fortes contraintes de temps dont la violation peut impliquer la violation des exigences de securité, de sûreté et de fiabilité.

Aujourd'hui, les STRse caracterisent par une forte intégration de composants logiciels. Leur développement nécessite une méthodologie permettant de relier, même à partir de la phase de conception, le comportement du système au niveau fonctionnel avec les aspects non fonctionnels qui doivent être ténus en compte dans la mise en oeuvre et à l'exécution, [53], [7], [51].

Dans cette thèse nous nous interessons au problème de l'ordonnancement qui est nécessaire pour assurer le respect des contraintes temporelles imposées par l'application lors de l'exécution. L'ordonnancement consiste à coordonner dans le temps l'exécution des différentes activités afin d'assurer que toutes leurs contraintes temporelles sont satisfaites. L'ordonnancement de systèmes temps réel critiques embarqués est essentiel non seulement pour obtenir des bonnes performances mais surtout pour garantir leur correct fonctionnement.

Cette thèse contribue dans deux aspects de STR:

- Dans le chapitre 3 on présente un modèle pour une classe de STRinspiré par le langage Java et nous développons, à partir de ce modèle, un algorithme d'attribution de priorités statiques basé sur la communication entre tâches. Cet algorithme est simple mais incomplet.
- Dans le chapitre 5 on présente une technique pour traiter le problème d'ordonnançabilité avec préemption, dépendences et incertitude. Nous etudions le problème d'analyse et decidabilité à travers d'une nouvelle classe d'automates temporisés.

Nous completons notre présentation avec un chapitre dévoué aux modèles temporisés, chapitre 4, et le chapitre 2 avec les techniques et méthodes d'ordonnançabilité les plus connus.

1.1 Real Time Systems

No doubt that computers are everywhere in our daily life. Some years ago, but not so many, computers were devices which had some "external" recognizable aspect, such as a box, a screen and a keyboard, generally used for calculating, data basing and business management. As communication, multimedia and networking were added to computing systems, the use of computers expanded to everyone; nowadays computers are integrated to planes, cars, multimedia systems and even... refrigerators!

A huge branch in computer systems began to develop, when computers were integrated to engins where *time* played a very important role. Any computer system deals with time, in a broad sense; in some systems, time is important because calculations are very heavy and the response time depends on the architecture of the system and the algorithm implemented, but time *is not* part of the system, that is, time is not part of the especification of the problem.

These systems are now wideley employed in many real time control applications such as avionics, automobile cruise control, heating control telecommunications and many other areas. The systems must also respond dynamically to the operating environment and eventually adapt themselves to new conditions; they are commonly called *embedded* since the "computing engine" is almost hidden and dedicated to the application.

Real Time Systems, RTS, deal with time in the sense that a response is demanded within a certain delay; if this demand is not satisfied, we could produce a failure, an accident and in general a *critical* situation. Compare, for instance the fact of using an ATM to withdraw money and a car airbag system; the first action takes some time, but the system does not deal with time; we can take some seconds to do the operation and even if the system is overcharged the user tolerates some unspecified delay (depending on his patience!); the airbag system deals with time, since its response, in case of an accident, must be given within a specified delay, if not, the driver could be hurt. Besides, a late response of the system is useless, since the consequences of the accident had already happened.

Even if a definition of RTS can lead to restrict ourselves, it is worth mentioning one:

> A real time system is a computing system where time is involved in the specification of the problem and in the response of the system. The correctness of computations depends not only on the logical correctness of the implementation, but also on the time response.

RTS can be classified into *hard* and *soft* RTS; in general, we say that in hard RTS the absence of an answer or an answer which fails to arrive on time can cause a critical event or unsafety situation to happen; in soft RTS even if the response deals with the time it is produced, the absence of an answer leaves the system in a correct state and some recovery can be possible. An example of soft RTS is the integration process while sending video frames; the system is quite time-dependant, in the sense that frames must arrive in order and also respect some timing constraints, to give the user the idea of viewing a "continous" film; but if eventually a frame is lost or if it arrives late, the whole system is correct and, principally, no critical event is produced.

The frontier between soft and hard RTS is sometimes not so clear; consider, for instance our example of video, it could be classified as *hard* if the "film" transmitted was a distance surgery operation. Sometimes, soft RTS are more difficult to specify since it is not easy to decide which timing requiremnts can be relaxed, and how they can be relaxed, how often and so on, [58].

As RTS deals with the "real world", the computer is dedicated to control some part of a system or physical engine; normally, the computer is regarded as a component of the piece to control and we say that these computer systems are *embedded*; sometimes, people are surprised to notice that nowadays a car has a computer in it, since the "traditional" view of a computer is not present. We really mean that a processor is installed, dedicated to survey a part of the system which interacts with the real world and offers an answer to a specific stimulus in a predetermined time. Airbag systems, ABS, heating and other "intelligent" household equipments, are examples of RTS which we do not see as such but are present in daily life. Airplane control, electronical control of trains and barriers, nuclear submarines have grown much safer since they were helped by computers.

In summary, RTS show some deep differences compared to traditional systems, [34]:

1. Time: in pure computing applications, time is not taken into account; one can talk about the *order* of an algorithm as a measure of *process time consuming* but time *is not* part of the algorithm. In RTS time must be modelled somehow and there are attempts to represent time in some temporal logics, [47], or in timed automata, [10, 22].

2. Events: in RTS the inputs can be considered as data under the form of events. These events are triggered by a sensor or by another (external) process, which we will generally call *producer*. On the other hand, these events are served by another process which we will call *consumer*. RTS are characterized by two basic sytles of design, [49], *event-driven* and *time-driven*. Time driven activities are those ruled by time, for instance, periodic activities, in which an event (task, in this case) is triggered simply by time passing. Event driven activities are those ruled by the arrival of an external event which may or may not be predicted; it fits reactive applications.

3. Termination: in the Turing-Church frame, computing is a terminating process, giving a result. A non terminating process is considered as defective. However RTS are intrinsically non terminating processes and even more, a terminating program is considered defective. In summary, in traditional applications ending is really **expected** but in RTS ending is **erroneous**.

4. Concurrency: even if some efforts have been done to manage concurrency and parallelism, the traditional criteria for software is based on the idea of *serializability*, which is perfectly embedded in the Turing-Church architecture. In RTS applications parallelism is the natural form of computation as a mechanism of modelling a *real life* problem, so we are faced to a scenario where multiple processes are running and interacting.

1.2 Traditional vs Real Time Software

Software development has dramatically changed since its beginings in the early 50. In those days, software was really wired to the computer, meaning that an application was in fact implemented for a given architecture; the simplest modification implied re-thinking all the application and re-installing the program.

Such a construction of software had no methodology; in the earlys 60 many programming languages were developed and a very important concept, *symbolic memory allocation* let programmers perform an abstraction between a program and a given architecture. Programs could be more or less exported or run into different machines: the concept of *portability* was born, but the activity of programming was reduced to the fact of knowing a language and *coding* an application in such a language.

At the end of 60's, the programming community realised that the situation was chaotic; programs were more and more important and large and the programming activity implied many people working over the same application. Besides that, it was clear that programming was much more than simply *coding*, implying at least three phases: *modelling*, *implementation* and *maintainance*.

The first phase is of most importance, since the application specification is clearly established and all actors involved in it express their views of the problem and their needs. Once we have such a plan of the application and that all restrictions are neatly written down, we can attack the second phase. The modelling phase has spread the problem into simpler components, with interaction among components so programmers can attack the implementation of components in parallel since they only need to know the "input" and "output" of each component, leaving the functional aspect of other components as a black box. Evidently the third phase is capital to the *evolution* of the application, as new needs may appear and if modelling is correct, we should only modify or create some few components but no need to re-implement the *whole* system.

The construction of RTS began by designing and implementing ad-hoc software platforms, which were not reusable for other applications; in this sense, RTS suffered the same experience as programming in the early 60's... no methodology was applied, and hence soon people were in face of the chaos which conducted to the development of good software design and analysis practices. No doubt that there has been a great shifting from hardware to software and hence we can now think of in terms of a "real time engineering", that is, based on some common models, we can use some previously developped (and proved) components or modules to build a new system, [34]. Of course, most embedded systems include a significant hardware design also, as new technologies are developped and a wider area of application is included.

The role of RT Modelling

Time is of most importance in RTS since we deal with critical applications, whose failure may cause serious or fatal accidents and also with different technologies which must be integrated. Building embedded RTS of guaranteed quality in a cost-effective manner, raises a challenge for the scientific community, [51].

As for any process of software construction, it is of paramount importance to have a good model which can aid the design of good quality systems and facilitate analysis and control.

The use of models can profitably replace experimentation on actual systems with many advantages such as facility to modify and play with different parameters, integration of heterogeneous components, observability of behaviour under different conditions and the possibility of analysis and predictability by application of formal methods.

The problem of modelling is to represent accurately the complexity of a system; a too "narrow" design could simplify the application to the point of being unreal; on the other hand, lack of abstraction conducts to a complexity which difficults the perception of properties and behaviour.

Modelling techniques are applied at early phases of system development and at high abstraction level. The existence of these techniques is a basis for rigorous design and easy validation.

A very important issue in real time modelling is the *representation of time* which is obtained by restricting the behaviour of its application software with timing information. Sifakis in [51] notes that a

> ... deep distinction between real time application software and the coresponding real time system resides in the fact that the former is immaterial and thus untimed

Time is external to the application and is provided by the execution platform while the operational aspects of the application are provided by the language; so the study of a RTS requires the use of a timed model which combines actions of the applications and time progress.

The existence of modelling techniques is a basis for rigorous design, but building models which faithfully represent RTS is not trivial; besides models are often used at an early stage of system development at a very high abstraction level, and do not easily last the whole design life-cycle, [52].

There are many different models of computation which may be appropriate for RTS such as actors, event based systems, semaphores, synchronization mechanism or synchronous reactive systems, [5, 4, 14, 35, 38, 39]. In particular, we have used finite state machines, FSM as a model; each machine represents a process, where the nodes of FSM represent the different states and the arcs the transitions or evolution of the process. Each process is then an ordered sequence of states ruled by the transitions.

Each arc is labelled by conditions.

FSM cannot express concurrency nor time, so to tackle these problems we have used *timed automata*, TA, which are FSM extended with clocks. In this scenario a RTS is represented as a collection of TA, naturally concurrent, where coordination is done through event triggering. This structure permits a formal analysis, using for example model checking to test safety, [26, 25, 43, 42, 41] or synthesis for checking schedulability. [8, 9, 7, 52].

The role of Time

Components in a RTS include concurrent tasks often assigned to distinct processors. These tasks may interact through events, shared memory or by the simple effect of time passing. From the point of view of component design we need some definitions to declare temporal properties; temporal logic, [47] is the classic representation of time.

Traditional software is verified by techniques dealing with the functional aspects of the problem and their implementation; we prove that the code really performs or behaves as specified by the model. These properties are *untimed*; in RTS we have to add another axe of verification, the *non functional* properties which deal with the environment and more precisely with *real time*. For instance, if we say "event a is followed by an event b triggered at most δ units of time afterwards", we mean that the interval between termination of a and begining of b should be smaller than δ units of time *measured in real time*.

Independently of the architecture of the system, non-functional properties are checked through a timed model of the RTS; this activity is called *timing analysis*; we can also take an approach guided through *synthesis* where we look for a correct construction using methods that help resolving some choices made during implementation.

Some steps in the transition from application software to implementation of RTS include:

1. Partition of the application software into parallel tasks; these components include concurrent tasks often assigned to distinct processors which interact through events, shared memory or by the simple effect of time passing. From the point of view of component design we need some definitions to declare temporal properties.

2. Usage of some techniques for resource management and task synchronization. This coordination may be due to many factors: temporal constraints, access to common resources, synchronization among events, and so on. A *scheduler* is in charge of this coordination.

3. The choice of adequate scheduling policies so that non-functional properties of the application are respected; for RTS one of the critical missions of the scheduler is to assure the *timeliness* of the activities, that is, the respect of the temporal constraints, which form part of the tasks, at the same level as other parameters or their functionality.

Synchronous and asynchronous RTS Two paradigms are used in the design of RTS: *synchronous* and *asynchronous* approach.

The synchronous paradigm assumes that a system interacts with its environment and its reaction is fast enough to answer before a new external event is produced; this means that environment changes occurring during the computation of a step are treated at the next step.

The asynchronous paradigm attacks a multi-tasking execution model, where independent processes execute at their own pace and communicate via a message passing system. Normally, we have an operating system which is responsible for scheduling all tasks so as to perform properly.

Both techniques have their inconvenients; the hypothesis for the synchronous paradigm is not easy to meet, modularity cannot be easily handled and the asynchronous paradigm is less predictable and hard to analyse, [52].

The role of the Scheduler

As a RTS application is composed of many tasks, some kind of coordination is necessary to direct the application to a good result. A *scheduler* is the part of a system which coordinates the execution of all these activities. Roughly speaking scheduling may be defined as the

> "activity of arranging the execution of a set of tasks in such a manner that all tasks achieve their objectives"

This definition, although very imprecise, gives an idea of the complexity of the problem. Coordination may be due to many factors: temporal constraints, access to common resources, synchronization among events, and so on. A scheduler does not coordinate the execution per se, but its relationships with other activities. As already mentioned, one of the critical missions of a RTS scheduler is to assure the *timeliness* of the activities.

The activity of *scheduling* was born when many tasks were run over a machine and the CPU had to be shared among the tasks, we talk about *centralized scheduling*. These tasks were basically independent programs, triggered by users (or even the operating system). The kernel of the operating system decides which task must be executed and assures (more or less) a fair policy of CPU distribution for all tasks; in this context, time is not part of a task's description, but only its functionality (given by the code) is important. Later on, when distribution was possible, due to communication facilities among computers, the activity of scheduling *distributed tasks* was a natural extension of the centralized approach.

Since a scheduler deals with tasks, it is time to define them precisely, but not formally:

> A *task* is a unit of execution, which is supposed to work correcty while alone in the system, i.e. a task is a verified unit of execution

A task is then correct and must be executed entirely, although its execution can be interrupted by the system and resumed later. A task has its own environment of execution (local variables and data structures) and perhaps some shared environment, whose correctness must be guaranteed by the execution platform, while local correctness is ensured by the task itself.

Tasks are normally grouped to perform one or more functions, constituting a *system* or application.

A *real time task*, RTT, is a task whose effects (given by its functionality) must be seen within a certain amount of time called *critical time*; its response is needed for another task to continue or for system performances and the absence of response or a late service can cause fatal accidents. The critical time for a task is called its *deadline*, that is a task must response before this limit. Deadlines are measured in units of times.

In summary, we can envision three main actors in a scheduled system:

1. The **processes**, (sometimes referred to as tasks or modules), in charge of performing independent actions in coordination with other tasks controlled by the scheduler.

2. The **scheduler** or the software (eventually hardware) controlling the operations and the coordination of a series of processes, which is basically a timed system which observes the state of the application and restricts its behavior by triggering controllable actions.

3. The **environment** or a series of uncontrollable actions, events, processes arrival or processes termination.

Two important issues in the development of RTS is *analysis* and *synthesis* of schedulers. Analysis is the ability to **check** the model of a system to decide if it is correct and if it respects the temporal contraints of *all* tasks in the application. Synthesis is the ability to **construct** an implementation model which respects the temporal contraints. Of course, both techniques points out to answer the same questions: "is a system schedulable?", and if so, "can we construct or check that our implementation is schedulable?"

The construction of scheduled systems has been sucessfully applied to some systems, for example, scheduling transactions in the domain of data bases or scheduling tasks in the operating system environment. In the area of RTS the existing scheduling theory is limited because it requires the system to fit into a schedulability criterion, generally to fit into a mathematical framework of the schedulability criteria. Such studies relax one hypothesis at a time, for instance tasks are supposed to be periodic, or only worst case execution times are considered.

1.3 Contributions

This thesis concentrates on the definition of techniques for task synchronization and resource management, as shown in step 2 in the previous section.

- Chapter 3 is devoted to the development of a model and its verification techniques for a real time program written in a Java-like language which uses synchronization primitives for communication and common resources. We show how an abstraction of the program can be analysed to verify schedulability and correct resource management.
- Chapter 5 is devoted to schedulability analysis and decidability;
  - We first show a proper and new technique to deal with the problem of preemptive scheduling and decidability under an asynchronous paradigm;
  - We show an evolutive application of this method starting from a very simple policy and finishing to a general scheduling policy;
  - In each step of this evolution we show that our method is decidable, that is, that its application can leave the system in a safe state and that this state can be reachable.
  - We also show a complete admission analysis that can be performed off line in case of a set of periodic tasks and on-line in case of an hybrid set of periodic and aperiodic tasks; in any case, the admission is the simple computation of a formula.

We complete our presentation with chapter 4 dedicated to timed models, where we show the basic model of timed automata, some of its extensions and applications to schedulability analysis. The most well-known techniques for schedulability of real time systems are developed in chapter 2.

1.4 Thesaurus

Here's a list of the abbrevations used in this document:

Abbrevation	Meaning
DPCP	Dynamic Priority Ceiling Protocol
DBM	Difference bound Matrix
EDF	Earliest Deadline First
EDL	Earliest Deadline as Late as Possible
ETT	Event Triggered Task
FSM	Finite State Machine
IIP	Immediate Inheritance Protocol
JSS	Job Shop Scheduling
lcm	Least Common Multiple
LIFO	Last In First Out
RMA	Rate Monotonic Analysis
PCP	Priority Ceiling Protocol
PIP	Priority Inheritance Protocol
RTS	Real Time Systems
RTT	Real Time Task(s)
SRP	Stack Resource Policy
SSP	Slack Stealing Protocol
SWA	Stopwatch Automaton
TA	Timed Automaton
TAD	Timed Automaton with Deadlines
TAT	Timed Automaton with Task
TBS	Total Bandwidth Server
UTA	Updatable Timed Automaton
WFG	Wait For Graphs

Chapter 2

Setting some order in the Chaos: Scheduling

Résumé

Ce chapitre à pour but d'introduire les concepts basiques d'ordonnancement développés depuis 1973; en commençant par les modèles les plus classiques nous finissons avec les modèles les plus récents.

Une application temps réels est modélisée par un ensemble de tâches $T = \{T_1, T_2, \ldots, T_n\}$, chaque tâche $T_i, 1 \leq i \leq n$ est characterisée par la paire (E_i, D_i), oú E_i est le *temps d'éxécution* de T_i et D_i est *l'echeance relative.* Eventuellement, on peut ajouter P_i, $E_i < D_i \leq P_i$, la *période* pour T_i, c'est à dire, l'intervalle de temps entre deux arrivées d'une tâche. Certaines tâches sont dîtes *événementielles* s'il existe un événement qui les déclenche; finalement certains auteurs considèrent d'autres paramêtres, telles que le *jitter*, précedence entre tâches, etc.

Dans l'application on peut utiliser de ressources en commun; ces ressources partagées sont accedées par un protocole spécial qui garantit la bonne utilisation; les tâches qui n'utilisent pas de ressources en commun sont appellées *independantes.*

On organise ce chapître selon la taxinomie souivante:

Ensemble de tâches	Nature	Gestion de Priorités	Exemple d'algorithme
Periodiques	Independentes	Statique	RMA
		Dynamique	EDF
	Dependentes	Statique	PIP, PCP, IIP
		Dynamique	DPCP
Periodiques et non-periodique	Independentes	Statique	SSP, EDL, TBS
	Dependent	Dynamique	TBS
Event Triggered	Independantes		ETT
Complex Constraints			2-EDF

2.1 Schedulers

We have already introduced the need of some coordination among tasks, under a RT scenario. We have defined a real time application as a collection of tasks, each of which has some temporal constraints and may interact with the environement through events. In this chapter, we will introduce formal representations of RTS, that is, abstractions of real life as simple (or not so simple!) models. We will cover more or less 30 years of efforts in the area; results shown in this section are general and show the big headlines; detailed descriptions can be found, of course, in the original papers[1].

A real time application may be characterized by a set $T = \{T_1, T_2, \ldots, T_n\}$ of real time tasks, RTT, which may be triggered by external events; each task T_i is characterized by its parameters $[E_i, D_i]$ where E_i is the execution time and D_i is the relative deadline. The execution time is constant for a task, like in a worst case execution environment and it is the time taken by the processor to finish it without interruptions. Deadline D_i is relative to the arrival time r_i of a task T_i, (sometimes called *release time*) and it is the time for T_i to finish; if a task T_i arrives at time r_i, the sum $D_i + r_i$ is called the *absolute deadline*.

RTT may be *periodic*, that is, they are supposed to arrive within a constant interval; normally periodic tasks respond to the fact that some applications trigger tasks regularly; in this case, we can extend our set of parameters by P_i, the period for each task, and tasks must be finished before the next request ocurrs; we need then that $E_i \leq D_i \leq P_i$. Some authors normally assume $D_i = P_i$, [37].

If task arrival is not predictable, we will say that a task is *aperiodic*. Some authors make the difference between a *semiperiodic task* and an *eventual* task; the former may arrive within a certain boundary of time, while the latter is really unpredictable. We will not make this difference.

In the context of periodic tasks, we can see that each periodic task T_i is an infinite sequence of instances of the same task; we normally note these instances as $T_{i,1}, T_{i,2}, \ldots$; $T_{i,1}$ arrives at time $r_{i,1} = \theta_i$, called its *origin*, and its absolute deadline is $d_{i,1} = \theta_i + D_i = r_{i,1} + D_i$; in general we can say that the absolute deadline of the k^{th} arrival of T_i is $d_{i,k} = r_{i,k} + D_i = \theta_i + (k-1) \times P_i + D_i$; in many contexts, $\theta_i = 0$).

The question is then how to manage the set T in order to satisfy all of its objetives, modelled as parameters; since we assume tasks are correct, achieving task objectives is reduced to finish its execution before its deadline and by extension all tasks in T. This is the activity of scheduling.

Definition 2.1 *A scheduling algorithm is a set of rules that determines the task to be executed at a particular moment.*

Many scheduling policies exist, based on parameters such as execution time, deadlines and periodicity. Based on a set T of tasks characterized by its parameters, we can differentiate schedulers acording to:

- Priority management: assignment of priorities to tasks is one of the most used techniques to enforce schedulability; we distinguish:
  - static or fixed: where T is analysed before execution and some fixed priorities are associated which are valuable at execution time and never changed.
  - dynamic: where some criteria is defined to create priorities at execution time, meaning that each time a task arrives, a priority is assigned, perhaps taking into account the active set of

[1] As far as possible, we will try to keep an homogeneous notation and so, symbols may differ from the original works

	Non-Preemptive	Preemptive
Static	easy to implement too restrictive	easy to implement liveliness
Dynamic	intelligent priority assigment less restrictive	relatively hard to implement costly but tend to optimum

Figure 2.1: Schedulers

tasks in the system; priorities of tasks might change from request to request, according to the environment.

- Scheduler strength: as schedulers rule the management of tasks, they have the power to interrupt a task; we distinguish:
  - non-preemptive: each task is executed to completion, that is, once a task is chosen to execute, it will finish and never be interrupted.
  - preemptive: a task may be interrupted by a higher priority task; the interrupted task is put in a sleeping state, where all of its environment is kept and it will be resumed some time later in the future.
- Nature of tasks:
  - Independent: tasks do not depend on the initiation or completion of other tasks and they do not share resources; each task is then 'autonomous' and can be executed since its arrival.
  - Dependent: the request for a task may depend on the execution of another task, perhaps due to certain data produced and consumed later or due to application requirements, such as shared resources which impose some method to access them.

Static schedulers are very easy to model and to be treated by the scheduling manager, but they are very restrictive, since they are not adaptive; dynamic schedulers may take into account the execution environment and evolution of the system. Preemption is a well known technique based on the idea that the arrival of a more urgent task may need to interrupt the currently executing one; this technique introduces another problem, *liveliness*, where a task shows no progress, since other (higher prioritized) tasks are continously delaying its execution.

Certainly, we can design schedulers based on a mixed of concepts, table 2.1 shows the results of a mixtures, assuming independent tasks.

The easiest schedulers are static and from the system point of view non-preemptive. Scheduler decisions are fixed at analysis time, where priorities are assigned and as no preemption is accepted, the system executes to completion, no need to keep environments. Of course, these are the most restrictive schedulers but the easiest to implement. One well known problem with schedulers is *priority inversion* where a lower priority task prevents a higher priority one to execute. Static non-preemptive schedulers suffer this problem, since an executing task cannot be interrupted and hence a recently arrived task with higher priority must wait.

Static schedulers with preemptions are very common; a newly arrived task interrupts the currently executing task if the later has lower priority than the former. The problem is that preemption introduces the problem of liveliness, since interrupted tasks may never regain the processor if higher priority tasks arrive constantly; some solutions have been proposed to this problem, specially the ceiling protocols, [50], [48].

Corresponding to the class of fixed priority schedulers, Rate Monotonic Analysis, RMA, is the most popular, [37].

Dynamic schedulers with no preemptions are not very common, because, in principle the "intelligence" of the assignment procedure is hidden by the incapacity of interruption; they are less restrictive than static but not sufficiently efficient.

Finally, dynamic schedulers with preemptions are the richest ones, the hardest to implement but the nearest to the optimum. Among the dynamic protocols, the most popular is the Earliest Deadline, ED, [37], and from this protocol, a very wide branch of algorithms exist.

From now on, we will use the following defintions for a schedule algorithm:

Definition 2.2 (Deadline Missing) *A system is in a* deadline missing *state at time t if t is the deadline of an unfinished task request.*

Definition 2.3 (Feasability) *A scheduling algorithm is* feasable *if the tasks are scheduled so that no deadline miss occurs.*

Definition 2.4 (Schedulable System) *A system is* schedulable *if a feasable scheduling algorithm exists.*

Feasability is the capacity of a policy or scheduling algorithm to find an arrangement of tasks to ensure no deadline missing, while schedulability is inherent to a set of tasks, that is, a set T may be schedulable even if the application of an algorithm leads to deadline missings. Finding whether a system is schedulable is much harder than deciding if it is feasable under a certain policy. We will show that certain algorithms ensures feasability for schedulable systems.

We organize this chapter following this taxonomy:

Task Set	Nature	Priority Management	Method Prototype
Periodic	Independent	Static	RMA
		Dynamic	EDF
	Dependent	Static	PIP, PCP, IIP
		Dynamic	DPCP
Periodic and Aperiodic	Independent	Static	SSP, EDL, TBS
	Dependent	Dynamic	TBS
Event Triggered	Independant		ETT
Complex Constraints			2-EDF

2.2 Periodic Independent Tasks

In this section, we show the main results in the area of scheduling RTT under the hypothesis that tasks are periodic and independent, i.e. each task is triggered at regular intervals or *rates*, it does not share resources and its execution is independent of other active tasks. We show two main classes of scheduler algorithms; the first class, **Rate Monotonic Analysis**, RMA, is based on static or fixed priority (generally attributed after an off-line analysis) and the second class, **Earliest Deadline First**, EDF, is based on dynamic priority assignment, based on the current state of the system.

2.2.1 Rate Monotonic Analysis

Rate Monotonic Analysis, RMA, was created by Liu and Layland in 1973, [37]. It is based in very simple assumptions over the set $T = \{T_1, T_2, \ldots, T_n\}$. Each task $T_i, 1 \leq i \leq n$ is characterized by its parameters $[E_i, P_i]$, for execution time and period, respectively, and it is assumed that:

- All hard deadlines are equal to periods.
- Each task must be completed before the next request for it occurs.
- Tasks are independent.
- Execution time is constant.
- No non-periodic tasks are tolerated for the application; those non-periodic tasks in the system are for initialization or recovery procedures, they have the highest priority and displace periodic tasks but do not have hard deadlines.

Definition 2.5 (Rate Monotonic Rule) *The rate-monotonic priority rule assigns higher priorities to tasks with higher request rates.*

A very simple way to assign priorities in a monotonic way is the inverse of the period. Priorities are then fixed at design time and schedulability can be analysed at design time. For a task T_i of period P_i its priority, π_i, is $\frac{1}{P_i}$.

The following theorem, due to Liu and Layland, [37] establishes the optimum criteria of RMA:

Theorem 2.1 *If a feasible priority assignment exists for some task set, the rate-monotonic priority assignment is feasible for that task set.*

An important fact in scheduling processing is the processor utilization factor, i.e, the time spent in the execution of the task set. Ideally, this number should be near to 1, representing full utilization of processor; but this is not possible, since there is some time in context switching and of course, the time used by the scheduler to take a decision.

In general, note that for a task T_i, the fraction of processor time spent in execution it is expressed by E_i/P_i, so for a set T of n task we have that the utilization factor U can be expressed as:

$$U = \sum_{i=1}^{n} (E_i/P_i)$$

This measure is slightly dependent of the architecture of the system, due to the "speed" E_i, but upper bounded by the deadlines which are architecture independent. Based on the utilization factor, Liu et al. established the following theorem:

Theorem 2.2 *For a set of n tasks with fixed priority assignment, the least upper bound for the processor utilization factor is $Up = n(2^{(1/n)} - 1)$.*

which in general shows an U in the order of 70%, rather costly in a real time environment. A better utilization bound it to choose periods such that any pairs shows an harmonic relation.

We show the application of RMA through an example due to [19].

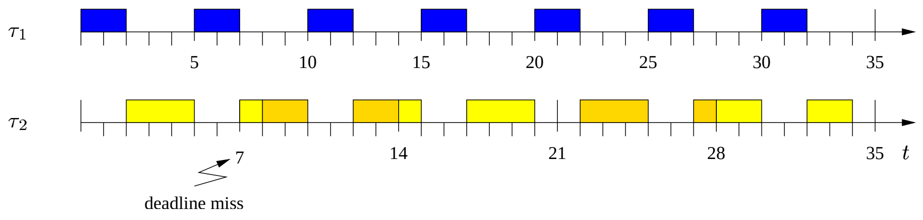


Figure 2.2: RMA application

Example 2.1 *Let us consider a set $T = \{T_1(2,5), T_2(4,7)\}$, of periodic tasks with parameters (E_i, P_i) as explained above, where periods are considered as hard deadlines. The utilization factor $U = \frac{2}{5} + \frac{4}{7} = \frac{34}{35}$; according to theorem 2.2 $Up \simeq 0.83$, so $U \geq Up$ and set T is not feasible, as shown in figure 2.2.*

As T_1 has a smaller deadline (or period) than T_2, T_1 has always the highest priority and preempts T_2 if it is executing, we can assign $\pi_1 = \frac{1}{5}$ and $\pi_2 = \frac{1}{7}$. The very first instance of T_2 misses its deadline since it is interrupted when the second instance of T_1 arrives; by time $t = 7$, when a second instance of T_2 arrives, it must yet complete the first instance, thus missing the deadline.

As priorities are fixed and known in advance, it suffices to analyse "a window" of execution between starting time, say $t = \theta$ and an upper bound called *hyperperiod*, $\mathcal{H}$ which is the least common multiple of the tasks periods.

Surely a better solution to RMA is a dynamic assignment algorithm. Liu et al. introduced a deadline driven scheduling algorithm called Earliest Deadline First, EDF.

2.2.2 Earliest Deadline First

This algorithm is based on the same idea as RMA, but in a dynamic way, i.e. the highest priority is assigned to the task with the shortest *current* deadline; it is based on the idea of urgency of a task. For performing this assignment we simply need to know the relative deadline of a task, P_i and its request time, r_i to calculate the absolute deadline.

For this algorithm the feasability is optimum in the sense that if a feasible schedule exists for a task set T, then EDF is also applicable to T.

Liu et al. established the following property for EDF:

Theorem 2.3 *For a given set of n tasks, the* EDF *algorithm is feasible if and only if*

$$U = \sum_{i=1}^{n} E_i / P_i \leq 1$$

which basically says that a set is feasible if there is enough time for each task, before its deadline expires.

Example 2.2 *Under this new policy, we can reconsider example 2.1, as $U = 0.97 \leq 1$ we know the set is schedulable, (the problem was that* RMA *was not feasable for that set). Figure 2.3 shows how considering absolute deadlines as a priority criteria enlarges the classes of schedulable sets.*

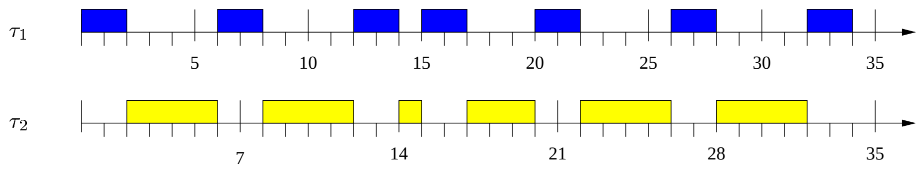


Figure 2.3: EDF application

- At time $t = 0$ both tasks arrive; $d_1 = 5 < d_2 = 7$ so T_1 starts.
- At time $t = 2$ T_2 gains the processor.
- At time $t = 5$ a new instance of T_1 arrives and absolute deadlines are analysed; for T_2 its absolute deadline is 7 while for T_1 is 10, so T_1 does not preempt T_2.
- At time $t = 6$ T_1 starts a new execution.
- At time $t = 7$ a new instance of T_2 arrives: $d_1 = 10 < d_2 = 14$, so T_2 waits. See how priorities have changed from one instance, to another.
- At time $t = 14$ T_2 arrives and begins execution.
- At time $t = 15$ T_1 arrives and $d_1 = 20 < d_2 = 21$ so T_1 preempts T_2.
- The rest of the instances is analysed analogously.

We now know that example 2.1 is schedulable even if RMA leaded to a deadline missing and we see that under certain conditions EDF is better than RMA.

2.2.3 Comparison

RMA is a fixed priority assignment algorithm, very easy to implement since at arrival of a new task T_i the scheduler knows whether it must preempt the currently executing task or simply accepting T_i somewhere in the ready queue. We assume that static analysis of the set T prevents the system to enter an unfeasable state.

EDF is a dynamic priority assignment algorithm which takes into account the absolute deadline $d_{i,k}$ of the k^{th} arrival of a periodic task T_i; in theory, this priority assignment presents no difficulty but in a system, priority levels are not infinite and there may be the case that no new priority level exists for a task. In this case, a complete reordering of the ready queue might be necessary.

Besides the natural consequence of calculating priorities at each task instance arrival, EDF introduces less runtime overhead, from the point of view of context switches, than RMA since preemptions are less frequent. Our example 2.1 shows this behaviour, see [19] for experimental results.

As seen by the theorems, a set of n tasks is schedulable by the RMA method if $\sum_{i=1}^{n} E_i/P_i \leq n(2^{1/n} - 1)$ while EDF extends the bound to 1. Some interestings results were shown if any pair of periods follows an harmonic relation. Under this hypothesis, RMA is also bounded to 1.

The most important result for EDF is that if a system is unschedulable with this method, then it is unschedulable with **all** other orderings. This is the *optimal* result for Liu et al.

One restriction under both protocols is that no resource sharing is tolerated, since priorities are based on deadlines and not on the behaviour of each task. In the next section, we will discuss one of the most popular methods for scheduling tasks which share resources: the *priority ceiling* family. Another restriction is that EDF does not consider the remaining execution time with respect to deadlines, to assign priorities, in order to minimize context switchings.

But the most important result of Liu's work is *simplicity*; their work, written in 1973, showed the way to follow theoretical results before implementation and by those days, where we can consider that some kind of chaos was installed in the real time community, their results showed that some order exists that rule this chaos.

2.3 Periodic Dependent Tasks

In this section, we consider some scheduling protocols which relax one of the conditions of RMA and EDF: tasks are independent. We consider algorithms where tasks *share* resources which are managed by the scheduler in a *mutually exclusive* way, that is, only one task at a time can access a resource; hence when a task demands a resource, it must wait if another task is using it.

Normally, resources are used in a *critical section* of the program and are accessed through a *demand* protocol, a task τ must *lock* a resource before using it, the system may *grant* it or *deny* it; in the latter case, τ must wait in a waiting queue, its execution is temporaly suspended still retaining other granted resources. This situation may cause a common problem: *deadlock* that is, a chain of tasks are suspended, each of which is waiting for a resource granted to another also suspended task. The systems shows no evolution through time.

Protocols shown in this section are called *deadlock preventive*, that is they prevent the situation where a deadlock is possible, by "guessing" somehow that in the future a deadlock will occur; to do this, they need some information, as the set of resources that a task may eventually access.

We present three classes of protocols based on *inheritance* of priorities assigned statically: **Priority Inheritance Protocol**, PIP, **Priority Ceiling Protocol**, PCP and **Immediate Inheritance Protocol**, IIP. Finally we present another protocol where priorities are managed dynamically: *Dynamic Priority Ceiling Protocol*, DPCP.

2.3.1 Priority Inheritance Protocol

The Priority Inheritance Protocols, [50], were created to face the problem of non-independent tasks, which share common resources. Each task uses binary semaphores to coordinate the access to critical sections where common resources are used and is assigned a priority (static or dynamic) which it uses all long its execution. Tasks with higher priorities are executed first, but if at any moment, a higher priority tasks T_i demands a resource allocated to a lower priority task T_j, this task *steals* or *inherits* the priority of T_i, thus letting its execution to be continued; after exiting the critical section, T_j returns to its original priority.

The original protocol assumes that:

1. Each task is assigned a fixed priority and all instances of the same task are assigned the same task's priority.

2. Periodic tasks are accepted and for each task we know its worst case execution time, its deadline and its priority.

3. If several tasks are eligible to run, that with the highest priority is chosen.
4. If several tasks have the same priority, they are executed in a first come first served, FCFS, manner.
5. Each task uses a binary semaphore for each resource to enter the critical section; critical sections may be nested and follow a "last open, first closed" policy. Each semaphore may be locked at most once in a single nested critical section.
6. Each task releases all of its locks, if it holds any, before or at the end of its execution.

Normally, a high-priority task T_i should be able to preempt a lower priority task, immediately upon T_i's initiation, but if a lower priority task, say T_j owns a resource demanded by T_i, then T_j is not preempted and even more, T_j will continue its execution even its low priority. This phenomenon is called *priority inversion* since a higher priority task is *blocked* by lower priority tasks and it is forced to wait for their completion (or at least for their resources).

The interest of the PIP is founded on the fact that a schedulability bound can be determined: if the utilization factor stays below this bound, then the set is feasable.

When a task T_i blocks one or more higher priority tasks, it ignores its original priority assignment and executes its critical section at the highest priority level of all the tasks it blocks. After exiting its critical section, task T_i returns to its original priority level.

Basically, we have the following steps:

Rule 1 The highest priority task is always executing except... Task T_i with the highest priority gains the processor and starts running. If at any moment T_i demands a critical resource r_j, it must lock the semaphore S_j on this resource. If S_j is free, T_i enters the critical section, works on r_j and on exiting it releases the semaphore S_j and the highest priority task, if any, blocked by task T_i is awakened. Otherwise, T_i is blocked by the task which holds the lock on S_j, no matter its priority.

Rule 2 No task can be preempted while executing a critical section on a granted resource r_j. Each task T_i executes at its assigned priority, unless it is in a critical section and blocks higher priority tasks; in this case, it inherits the highest priority of the tasks blocked by T_i. Once T_i exits a critical section, the scheduler will assign the resource to the highest priority task demanding r_j. This is very important in nested levels; consider a task T_i which includes code like this:

```
...
lock(r1)
...
lock(r2)
...
unlock(r2)
...
unlock(r1)
...
```

Once the task T_i releases r_2 it regains the priority it had *before* locking r_2; this may be lower than its current priority and T_i may be preempted by the task with the highest priority (perhaps one blocked by T_i but not necessarily). Of course, T_i still holds the lock on r_1, with the priority assigned for the highest priority task which had demanded r_1.

Rule 3 Priority inheritance is transitive. As a consequence of the previous observation, we deduce that inheritance is transitive. We show this through an example:

Example 2.3 (Inheritance of Priorities) *Imagine three tasks T_1, T_2 and T_3 in descending priority order. If T_3 is executing then it blocks T_2 and T_1 as it owns a common resource wanted by T_2 or by both tasks (if not, T_3 could not be executing). Task T_3 inherits the priority of T_1 via T_2.*

Consider the following escenario:

```
Task T3              Task T2              Task T1
...                  ...                  ...
lock(a) (1)          lock(c)   (2)        lock(d)   (3)
...                  ...                  ...
lock(b) (6)          lock(a)   (5)        lock(c)   (4)
```

The numbers between brackets indicate the order of execution. T_3 starts execution and locks a, then T_2 enters the system and preempts T_3 as its priority is higher (for the instant being, T_3 has not yet inherited T_2's priority) accessing the critical section for resource c. Then T_1 gains the processor as it has the highest priority accessing d and it intends to lock resource c which is owned by T_2 (point (4)); *at this moment* T_2 inherits T_1's priority, resumes its execution until point (5) where resource a is owned by T_3 and *at this moment* this task inherits T_2's priority which is, in fact, T_1's one.

Rule 4 Highest priority task first. A task T_i can preempt another task T_j if T_i is not blocked and its priority is higher than the priority, *inherited or assigned*, at which T_j is running.

This protocol has a number of properties; one of the most interesting is the fact that a task T_i can be blocked for at most the duration of one critical section for each task of lower priority. Although we do not give the proof, the example shown above is illustrative of this fact.

As a consequence of this mechanism, the basic protocol *does not prevent deadlocks*. It is very easy to see through this example:

Example 2.4 (Deadlock)

```
Task T2              Task T1
...                  ...
lock(a)   (1)        lock(b) (2)
...                  ...
lock(b)   (4)        lock(a) (3)
...                  ...
unlock(b)            unlock(a)
...                  ...
unlock(a)            unlock(b)
```

where T_1 has highest priority. T_2 enters the systems (1) locking a, then T_1 at (2) preempts T_2, locks b and when it intends to lock a (3) is blocked by T_2, which regains the processor (as it inherits the priority of T_1); when T_2 intends lock b (4) this resource had already been assigned. Both tasks are mutually blocked, hence in deadlock.

This problem can be faced by imposing a total ordering on the sempahore accesses, but blocking duration is still a problem since a chain of blocking can be formed as showned in the examples above.

2.3.2 Priority Ceiling Protocol

Priority Ceiling Protocol, PCP, is a variant of the basic PIP but it prevents the formation of deadlocks and chained blocking. The underlying idea of this protocol is that a critical section is executed at a priority higher than that of any task that could *eventually* demand the resource. **The PIP promotes an ascending priority assignment as new higher piority tasks enters the systems and are blocked by lower priority tasks, but the PCP assigns the highest priority to the task which first gets the resource among all active tasks demanding the resource**.

To implement this idea, a priority ceiling is first assigned to each semaphore, which is equal to the highest priority task that could *ever* use the semaphore. We accept a task T_i to begin execution of a new critical section if its priority is higher than all priority ceilings of all the semaphores locked by tasks other than T_i. Note that the *demanded* resource is not taken into account to access the critical section, but the *ceilings* of other active tasks.

Let us revisit our example 2.4 to see how it works:

Example 2.5 (Deadlock Revisited) *Initially T_2 enters the system and locks resource a (1); later, T_1 enters the system, preempts T_2 and when it tries to lock b (which is free), the scheduler finds that T_1's priority is not higher than the priority ceiling of the* locked *semaphore a; T_1 is suspended and T_2 resumes execution; when T_2 tries to lock b it has in fact* the highest *priority since no other tasks locks a semaphore; hence, T_2 locks, executes, finishes and releases all of its resources, letting T_1 continue its execution. Observe that even when T_2 releases b, the scheduler will not let T_1 resume its execution, since its priority is* still *lower than T_2's.*

The protocol can be summarized in the following steps:

Step 1 A task T_i with the highest priority is assigned to the processor; let S^* be the semaphore with the highest priority ceiling of all semaphores currently locked by tasks other than T_i. If T_i tries to enter a critical section over a semaphore S it will be blocked if its priority is not higher than the priority ceiling of semaphore S^*. Otherwise T_i enters its critical section, locking S. When T_i exits its critical section, its semaphore is released and the highest priority task, if any, blocked by T_i is resumed.

Step 2 A task executes at its fixed priority, unless it is in its critical section and blocks higher priority tasks; at this point it inherits the highest priority of the tasks blocked by T_i. As it exits a critical section, it regains the priority it had just before entry to the critical section.

Step 3 As usual, the highest priority task is always executing; a task T_i can preempt another task T_j, if its priority is higher than the priority at which T_j is running.

Example 2.6 *Consider three tasks T_0, T_1 and T_2 in descending priority order, accessing resources a, b and c. We schematize the steps:*

```
Task T0             Task T1              Task T2
...       (5)       ...       (2)        ...
lock(a)   (6)/(9)   lock(c)   (3)/(12)   lock(c)    (1)
...                 ...                  ...
unlock(a)           unlock(c)            lock(b)    (4)
...                 ...                  ...        (7)
lock(b)                                  unlock(b)  (8)
...                                      ...        (10)
unlock(b)                                unlock(c)  (11)
...                                      ...        (13)
```

The priority ceilings of semaphores for a and b are equal to T_0's priority and for c at T_1's priority. Figure 2.4 illustrates the sequence of events.

- At time t_0 task T_2 begins its execution and blocks c (1).
- At time t_1 task T_1 enters the system, (2), preempts T_2 and begins its execution but it is blocked when it tries to lock c (3) owned by T_2, which resumes its execution at T_1's priority (inheritance).
- At time t_2, task T_2 enters its critical section for b since no other semaphore is locked by other jobs (4).
- At time t_3, task T_0 enters the system, (5), and as it has a higher priority, it preempts T_2, which is still in b's critical section; note that T_2's priority (in fact, inherited from T_1), is lower than T_0's.
- At time t_4 as T_0 tries to enter the critical section for a, (6), it is blocked since its priority is not higher than the priority ceiling of the locked semaphore for b. At this point, T_2 regains the processor at T_0's priority (inheritance), (7).
- At time t_5, T_2 releases the semaphore for b, (8), and returns to the previously inherited priority from T_1 but T_2 is preempted by T_0 which regains the processor, (9).
- At time t_5, T_0 accesses the critical section for a and it is never stopped until termination since it has the highest priority.
- At time t_6, T_2 resumes its execution, (10), at T_1's priority, exits the critical section for c, (11), recovers its original priority and is preempted by T_1.
- At time t_7, T_1 is granted the lock over c, (12), finishes its execution (time t_8) and then T_2 resumes, (13), and also terminates (time t_9).

Many properties characterize this protocol: it is deadlock free and a task will not be blocked for more time than the duration of one critical section of a lower priority task; it also offers a condition of schedulability based on a RMA assignment of priorities for a set of periodic tasks:

Theorem 2.4 (Schedulability of PCP) *A set of n periodic tasks using the PCP can be scheduled by the RMA if the following condition is satisfied:*

$$\sum_{i=1}^{n} \frac{E_i}{P_i} + max\left(\frac{B_1}{P_1}, \ldots, \frac{B_{n-1}}{P_{n-1}}\right) \leq n(2^{1/n} - 1)$$

where B_i is the worst case blocking time for a task T_i, that is, the longest duration of a critical section for which T_i might eventually wait.

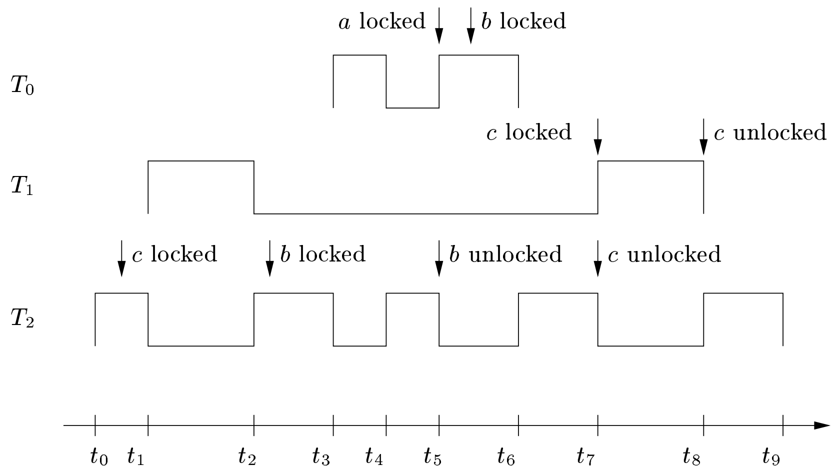


Figure 2.4: Sequence of events under PCP

2.3.3 Immediate Inheritance Protocol

The main difficulty with PCP is implementation in practice, since the scheduler must keep track of which task is blocked on which semaphore and the chain of inherited priorities; the test to decide whether a semaphore can be locked or not is also time consuming.

There is a very simple variant of this method, called *immediate inheritance protocol*, IIP, which indicates that if a task T_i wants to lock a resource r, the task immediately sets its priority to the maximum of its current priority and the ceiling priority of r. On exiting the critical section for r, T_i comes back to the priority it had just before accessing r.

Each task is delayed at most once by a lower priority task, since there cannot have been two lower priority tasks that locked two semaphores with ceilings higher than the priority of task T_i, since one of them would have inherited a higher priority first. As it inherits a higher priority, the other task cannot then run and lock a second semaphore. One of the consequence of this protocol is that if a task T_i is blocked, then it is blocked *before* it starts running, since if other task T_j is running and holds a resource ever needed by T_i then it **has** at least T_i's priority; so when task T_i is released it will not start running until T_j has finished.

This variation of the PCP is easier to implement and can be found in many commercial real time operating systems, [58].

2.3.4 Dynamic Priority Ceiling Protocol

In this section we present a ceiling protocol which works dynamically; in PIP and all of its extensions, priorities are assigned statically: each task has a static priority and each resource has a ceiling priority which varies from PIP to PCP. Each task changes dynamically its priority as it demands resources but it always starts at the same priority, regardless of the environment. We have shown the schedulability result under the static assignment for RMA.

The *Dynamic Priority Ceiling Protocol*, DPCP, was created by Chen et al in [21] and extended by Maryline Silly in [54]. A task T_i is assigned a dynamic priority according to EDF protocol; as usual a task T_i may lock and unlock a binary semaphore according to a PCP. A priority ceiling is defined for every critical section and its value at any time t is the priority of the highest priority task, (the task with the earliest deadline), that may enter the critical section at or after time t.

Each release of T_i may be blocked for at most B_i, the worst case blocking time. B_i corresponds to the duration of the longest critical section in the set $\{s, s \in S_j \cap S_k, D_k \leq D_i < D_j\}$, where s is a semaphore to access a resource and S_i is the list of semaphores accessed by T_i.

A very simple sufficient condition for the set T to be schedulable is

$$\sum_{i=1}^{n} \frac{E_i + B_i}{P_i} \leq 1$$

in which we "add" to the normal worst case execution time, the blocking time, assuming it as an extra computation. We need a more precise schedulability condition for DPCP.

We will assume that deadline equals periods and we define the *scheduling interval* for a request T_i to be the time

$$[r_i, f_i]$$

where r_i is the release time and f_i is the completion time for T_i. We will denote c_j as the deadline associated to the ceiling priority of sempahore S_j, in fact, c_j is the deadline of the highest priority task that uses or will use semaphore S_j.

Let I_i be a scheduling interval for T_i in which the maximal amount of computation time is needed to complete T_i and all higher priority tasks. Of course there may be a lower priority task that can block T_i in I_i; let m be the index of this task. Let $\mathcal{L}_i$ be the ordered set of requests' deadlines within the time interval $[D_i, D_m]$ and let $L_i = \min_{t \in \mathcal{L}_i}(t - \sum_{j=1}^{n} \lfloor \frac{t+x_j}{P_j} \rfloor . E_j)$. L_i represents a lower bound of additional computation time that can be used within I_i while guaranteeing deadlines of lower prioriy tasks.

Theorem 2.5 (Silly99) *Using a dynamic* PCP *all tasks of* T *meet their deadlines if the two following conditions hold:*

$$\sum_{i=1}^{n} \frac{E_i}{P_i} \leq 1 \tag{2.1}$$

$$B_i \leq L_i \, \forall i, 1 \leq i \leq n \tag{2.2}$$

See [54] for proof.

Example 2.7 *Consider three tasks* $T_1 = (4, 12, 16), T_2 = (6, 20, 24), T_3 = (8, 46, 48)$, *where the first parameter represents execution time, the second the deadline and the third the period. Analysis is done within the interval* $[0, 48]$ *where three instances of* T_1, *two of* T_2 *and one for* T_3 *will arrive.* T_1 *accesses semaphore* S_1, T_2 *accesses* S_2 *and task* T_3 *both of them.* S_1 *takes 2 units to be unlocked and* S_2 *takes 4. Conditions 2.1 and 2.2 are satisfied; according to deadlines, task* T_1 *has the highest priority and hence* S_1 *and* T_3 *has the lowest;* S_2 *is assigned* T_2*'s priority. Figure 2.5 shows the schedule produced by a dynamic* PCP *using the* earliest deadline as late as possible, EDL, *which promotes pushing the execution of periodic tasks as late as possible, respecting their deadlines.*

- At time $t = 0$ the three tasks arrives: $d_1 = 12$, $d_2 = 20$ and $d_3 = 46$; T_1 is executed first at $t = 8$, the latest possible time to complete.

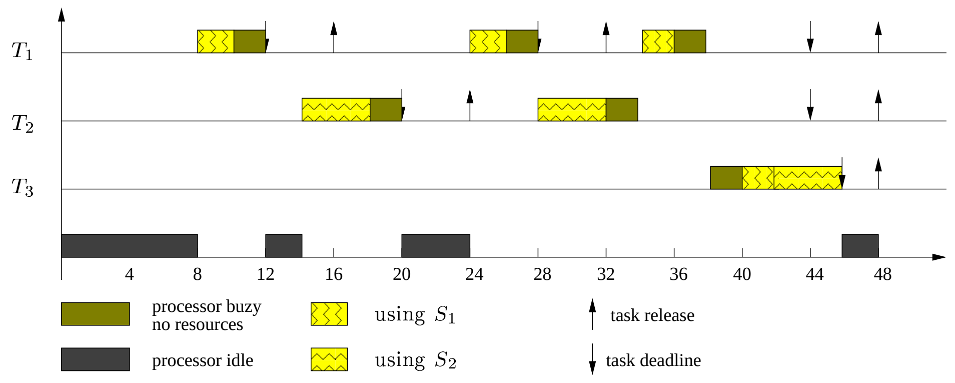


Figure 2.5: Dynamic PCP

- At time $t = 14$, T_2 is started following the EDL policy.
- At time $t = 16$, while T_2 is executing, a new instance of T_1 arrives, its deadline $d_1 = 28 > d_2 = 20$, T_1 does not preempt T_2.
- At time $t = 20$, T_2 completes and at $t = 24$ a new instance of T_2 arrives, $d_2 = 44$
- At time $t = 24$, T_1 starts and finishes at $t = 28$.
- As T_3 and T_2 latest starting time is 38 but T_2 deadline is 44, we start at $t = 28$ T_2.
- At time $t = 32$ while T_2 is executing the last instance of T_1 arrives with deadline $d_1 = 44$, so it does not preempt T_2.
- At time $t = 34$, T_1 starts and finishes at $t = 38$ unlocking resources ofor T_3 to start.

We will see in detail this algorithm in section 2.4.1.

2.4 Periodic and Aperiodic Independent Tasks

Our previous sections were dedicated to the problem of scheduling a set of periodic tasks; even if the methods can be extended to a mixture of periodic and aperiodic tasks, the main results over schedulability and bounded blocking time are found for a set of periodic tasks. In this section we will try to analyse some approaches to handle a mixture of periodic and aperiodic tasks.

In principle we define an *aperiodic* task as a unit of execution which has irregular and unpredictable arrival times, that is, a task that may be driven by the environment at any moment with no relation among arrivals. These kind of tasks may be executed as soon as possible after their arrival while periodic tasks might be completed later within their deadlines, taking advantage of the fact that we know their periodicity to push their execution as late as possible, but finishing before deadlines. In summary, we are respecting deadlines for periodic tasks and responsiveness for aperiodic tasks.

2.4.1 Slack Stealing Algorithms

In [36] and [57] we find a *slack stealing protocol*, SSP, which became the reference in scheduling a mixed set of tasks. The idea is to use the idle processor time to execute aperiodic tasks.

As usual, a periodic task T_i is characterized by its worst-case execution time, E_i, its deadline D_i and its period P_i, $D_i \leq P_i$; a task is initiated at time $\theta_i \geq 0$; periodic tasks are scheduled under a fixed priority algorithm, such as RMA, and by convention tasks are ordered in priority descending order.

For each aperiodic task J_i, we associate an arrival time α_i and a computing time of c_i. Tasks are indexed such that $0 \leq \alpha_i \leq \alpha_{i+1}$; between the interval $[0, t]$ we define the cumulative workload caused by executing aperiodic tasks:

$$W(t) = \sum_{i|\alpha_i \leq t} c_i$$

Any algorithm for scheduling both periodic and aperiodic tasks accumulates the effective execution time destinated to aperiodic tasks, $\epsilon(t)$, for a period $[0, t]$; of course $\epsilon(t) \leq W(t)$ which is an upper bound of execution times for aperiodic tasks.

Aperiodic tasks are processed in a FIFO manner; the completion of a task J_i, denoted by F_i is given by

$$F_i = \min\{t|\epsilon(t) = \sum_{k=1}^{i} c_k\}$$

and the response time for J_i, denoted R_i is given by

$$R_i = F_i - \alpha_i$$

The scheduling algorithm proposed by Lehoczky and Ramos-Thuel minimizes R_i, which is equivalent to minimize F_i.

The SSP uses a function $A_i(t)$ for each task T_i which represents the amount of time that can be allocated to aperiodic tasks within the interval $[0, t]$ which should run at a priority level i or higher, being the processor constantly busy and all tasks meeting their deadlines. The total amount of free time is $A(t)$; since tasks T_i's are periodic, it suffices to analyse the interval $[0, \mathcal{H}]$, where $\mathcal{H}$ is the least common multiple of the task periods.

1. For each periodic task T_i and for each instance j of T_i within $[0, \mathcal{H}]$ we compute

   $$\min_{(0 \leq t \leq D_{ij})}\{(A_{ij} + E_i(t))/t\} = 1$$

   which gives the largest amount of aperiodic processing possible at level i or higher during interval $[0, F_{ij}]$ such that $F_{ij} \leq D_{ij}$ (F_{ij} is the completion time for the j-th instance of task T_i),

2. At run time there are three different kind of activities: activity 0 is aperiodic task processing, activities $1 \ldots n$ is periodic task processing and activity $n + 1$ refers to the processor being idle.

3. At any time, we keep $\mathcal{A}$ the total aperiodic processing and $\mathcal{I}_i$ the i-level inactivity. We suppose periodic tasks are schedulable (by some other mechanism such as RMA). Suppose we start an activity j at time t, which finishes at time t' ($t' > t$) and $0 \leq j \leq n + 1$). Then if $j = 0$ we add $t' - t$ to $\mathcal{A}$ and if $2 \leq j \leq n$ then we add $t' - t$ to $\mathcal{I}_1, \ldots, \mathcal{I}_{j-1}$

4. When a new aperiodic task J arrives, we must compute the availability for this task. We compute

$$A^*(s,t) = min_{(1 \leq i \leq n)}(A_i(s,t)$$

and

$$A_i(s,t) = A_{ij} - A(s) - \mathcal{I}_i(s))$$

Suppose J arrives at time t with a w computing time; if $A^*(s,t) \geq w$ then we can process immediately at $[s, s+w]$, at the highest priority level (since we are preempting the currently executing task). If $A^*(s,t) \leq w$ then, we will execute at tiem $[s, s+A^*(s,t)]$ but no further aperiodic processing is available until additional slack; this will occur when a periodic job is completed.

The SSP is optimal in the sense that under a fixed priority scheduler for periodic tasks and a FIFO management for aperiodic tasks, the algorithm minimizes the response time for aperiodic processing among all scheduling algorithms which are feasible.

Calculating Idle Times

M. Silly, [54] introduced a very clear method to calculate static idle times for a set of independent periodic tasks; these idle times are used to compute aperiodic tasks. The analysis is based, as for Thuel's and Lehockzy's algorithm, on the assumption that periodic tasks may be executed as late as possible, (based on their deadlines), and that aperiodic tasks are executed as soon as possible. This algorithm is called *Earliest Deadline as Late as possible*, EDL.

We need to construct two vectors in the interval $[0, \mathcal{H}]$:

1. $\mathcal{K}$, called *static deadline vector*, which represents the times at which idle times occur and is constructed from the distinct deadlines of periodic tasks:

$$\mathcal{K} = (k_0, k_1, \ldots, k_i, k_{i+1}, \ldots, k_q)$$

where $k_i < k_{i+1}$, $k_0 = 0$ and $k_q = \mathcal{H} - min\{x_i, 1 \leq i \leq n\}$ where $x_i = P_i - D_i \, \forall 1 \leq i \leq n$.

2. $\mathcal{D}$, the *static idle time vector*, which represents the lengths of the idle times:

$$\mathcal{D} = (\Delta_0, \Delta_1, \ldots, \Delta_i, \Delta_{i+1}, \ldots, \Delta_q)$$

where each Δ_i gives the length of the idle time interval, starting at k_i, $1 \leq i \leq q$. This vector is obtained by the recurrent formula:

$$\Delta_q = \min\{x_i, 1 \leq i \leq n\} \quad (2.3)$$

$$\Delta_i = \max(0, F_i) \text{ for } i = q-1 \text{ down to } 0 \quad (2.4)$$

with $F_i = (\mathcal{H} - k_i) - \sum_{j=1}^{n} \lceil \frac{\mathcal{H} - x_j - k_i}{P_j} \rceil E_j - \sum_{k=i+1}^{q} \Delta_k$

Example 2.8 *Reconsider example 2.7. In principle $q = 6$ (or smaller); from formulae 2.3 and 2.4 we know that $k_0 = 0$ and $k_6 = 48 - min\{4, 4, 2\} = 46$ and $\Delta_6 = 2$. The 'last' moment to start running can be derived from the differences among deadlines and execution times. For T_1 this moment is at*

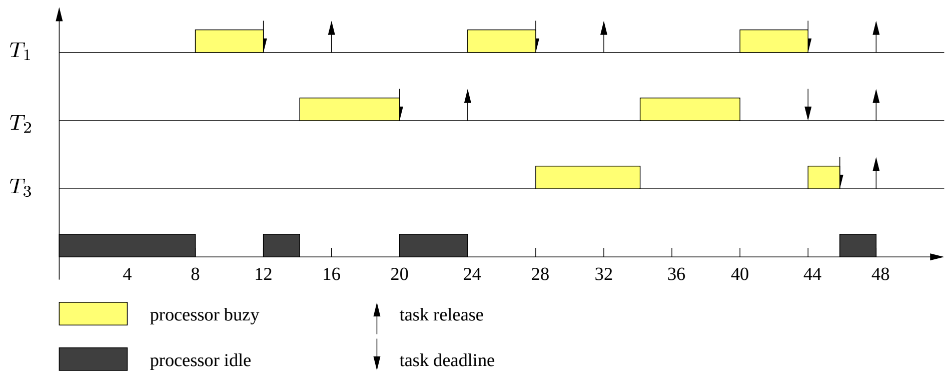


Figure 2.6: EDL static scheduler

8 (12-4); for T_2 is at 14 (20-6) and finally for T_3 is at 38 (46-8). Deadline vector $\mathcal{K}$ is constructed from deadlines; for T_1 these are 12, 28, and 44; for T_2 we have 20 and 44 and finally for T_3 we have 46; sorting these numbers gives $\mathcal{K} = (0, 12, 20, 28, 44, 46)$. Calculating $\mathcal{D}$ is a little more difficult. For instance,

$$F_5 = (48 - 44) - \sum_{j=1}^{3} \lceil \frac{48 - x_j - k_5}{P_j} E_j - \sum_{k=6}^{6} \Delta_k$$

which gives

$$F_5 = 4 - [0 + 0 + 8] - 2 = -6$$

so $\Delta_5 = 0$, and so on. Figure 2.6 gives the whole static scheduling

This information is now useful at processing time while a new aperiodic task task arrives. Suppose J arrives at time α with an execution time of E and a deadline D. We need to find an interval $[\alpha, \alpha + D]$ where at least E units of idle time exists, and this can be done easily by using our vectors $\mathcal{K}$ and $\mathcal{D}$, shifting the origin to α.

We will not give the details of these implementation, but only a simple example; see [54] for a full description and proofs.

Example 2.9 *Suppose at time $\alpha = 7$ a task J arrives with $E = 5$ and $D = 15$. We need to know if within $[7, 22]$ there exists 5 free units. We calculate this by creating $\mathcal{K}' = (7, 12, 20, 28, 44, 46)$ and $\mathcal{D}' = (1, 2, 4, 0, 0, 2)$. We have 1 unit in [7,8], 2 in [12,14] and 4 in [20,24], within [7,22] we have our 5 units. Task J may be accepted and vectors $\mathcal{K}$ and $\mathcal{D}$ must be corrected.*

Silly, [54] also proposed a dynamic algorithm to calculate idle times while using a dynamic priority algorithm for periodic tasks, such as EDF.

2.5 Periodic and Aperiodic Dependent Tasks

The model presented in this section, considers schedulability under a set of periodic and aperiodic tasks which share some resources.

Under this assumption, we cannot break an aperiodic task in multiple chunks to be executed in idle processor time, because tasks are now not independent and could affect the static schedulability for periodic tasks; on the other hand, if we schedule share resources by means of PCP, we need to assign to aperiodic tasks a deadline in order to create their priority. We will show a simple method, called Total Bandwidth Server, TBS, due to Spuri and Buttazzo, [55], [56] which assigns deadlines to aperiodic tasks in order to improve their responsiveness and manage common resources.

2.5.1 Total Bandwidth Server

The Total Bandwidth Server, TBS, improves the response time of soft aperiodic tasks in a dynamic real-time environment, where tasks are scheduled according to EDF. As usual, periodic tasks are characterized by their execution times and deadlines; aperiodic tasks are only characterized by their execution time. This protocol does *not* consider common resources but introduces some ideas which are used for a mix of periodic and aperiodic dependant tasks.TBS can be used for a set of periodic and aperiodic independant tasks.

We need a dealine for aperiodic tasks. When the k^{th} aperiodic request arrives at time $t = r_k$, it receives a deadline

$$d_k = \max(r_k, d_{k-1}) + \frac{C_k^a}{U_s}$$

where C_k^a is the execution time of the request and U_s is the server utilization factor. By definition $d_0 = 0$ and the request is inserted into the ready queue of the system and scheduled by EDF, as any (periodic) instance.

Example 2.10 *Consider two periodic tasks $T_1 = (3, 6)$ and $T_2 = (2, 8)$, where the first component represents execution time and the second the relative deadline (equal to period), see figure 2.7.*

Under this scenario, $U_p = \frac{3}{4}$ and consequently $U_s \leq \frac{1}{4}$. At time $t = 6$ while the processor is idle, an aperiodic task J_1 with $C_1 = 1$ arrives and its deadline is set to $d_1 = r_1 + \frac{C_1}{U_s} = 6 + \frac{1}{0.25} = 10$. Task can be scheduled since we are not exceeding the utilization factor, ($\frac{1}{10} < \frac{1}{4}$, and its deadline is the shortest (no other tasks are in queue), J_1 is served inmediately. We also show a task J_2 with $C_2 = 2$ which arrives at time $t = 13$ and is served at $t = 15$, since its deadline is set to 21 but a shorter deadline task is still active. Finally there is a task J_3 with $C_3 = 1$ which arrives at $t = 18$, executed at $t = 22$.

Actually, as can be seen in figure 2.7, TBS is not optimal, since we could improve the responsiveness of aperiodic jobs. The authors propose an optimal algorithm, called TB*, which iteratively shortens the assigned TBS deadline using the following property:

Theorem 2.6 (Buttazzo and Sensini,97) *: Let σ be a feasible schedule of task set T, in which an aperiodic task J_k is assigned a deadline d_k, and let f_k be the finishing time of J_k in σ. If d_k is substituted with $d'_k = f_k$, then the new schedule σ' produced by EDF is still feasible.*

2.5.2 TBS with resources

The duration of critical sections must be taken into account when we handle common resources. In fact, when we have a mixture of periodic and aperiodic tasks, we need to bound the maximum blocking time of each task and analyse the schedulability of the hybrid set at arrival of a new aperiodic job. Buttazzo et al. based their algorithm assuming a Stack Resource Policy, SRP, [11], to handle shared resources. We describe briefly this policy.

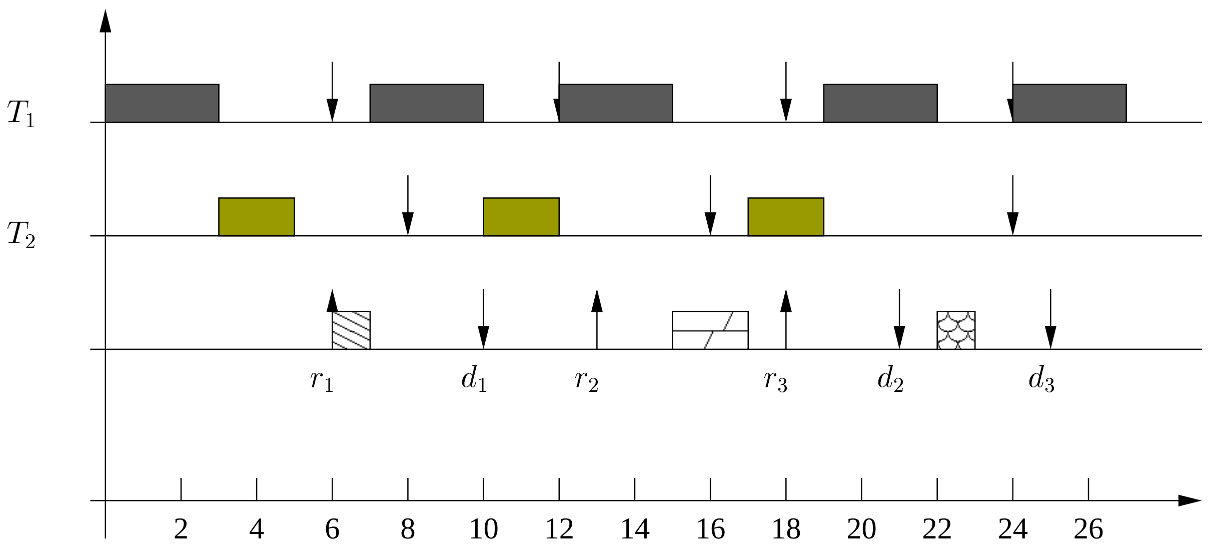


Figure 2.7: TBS example

In the TBS with resources, every task T_i is assigned a dynamic priority p_i based on EDF and a static preemption level π_i such that the following property holds:

Property 2.1 (Stack Resource Policy) *Task T_i is not allowed to preempt task T_j, unless $\pi_i > \pi_j$.*

The static priority level for a task T_i with relative deadline D_i is $\pi_i = \frac{1}{D_i}$. In addition, every resource R_k is assigned a ceiling defined as:

$$\text{ceil}(R_k) = \{\pi | T_i \text{ needs } R_k\}$$

Finally a dynamic system ceiling is defined as:

$$\Pi_s(t) = \max[\{\text{ceil}(R_k) | R_k \text{is currently busy } \} \cup \{0\}]$$

The SRP rule states that:

> "a task is not allowed to start executing until its priority is the highest among the active tasks, noted act(T), and its preemption level is greater than the system ceiling".

That is, an executing task will never be blocked by other active tasks though it can be preempted by higher priority tasks but no blocking will occur.

Under this protocol, a task never blocks its execution; it cannot start executing if its preemption level is not high enough; however, we consider the time waiting in the ready queue as a blocking time B_i since it may be caused by tasks having lower preemption level. The maximum blocking B_i for task T_i can be computed as the longest critical section among those with a ceiling greater than or equal to the preemption level of T_i, (a similar reasoning have been applied in [54]):

$$B_i = \max_{(T_j \in act(T))}\{s_{j,h} \mid (D_i < D_j) \wedge \pi_i \leq \text{ceil}(\rho_{j,h})\} \tag{2.5}$$

where $s_{j,h}$ is the worst case execution time of the h^{th} critical section of task T_i and $\rho_{j,h}$ is the resource accessed by the critical section $s_{j,h}$.

The following condition:

$$\forall i, 1 \leq i \leq n \sum_{k=1}^{i} \frac{E_k}{P_k} + \frac{B_i}{P_i} \leq 1 \tag{2.6}$$

can be tested to ensure feasibility of a set of periodic tasks with common resources.

To use SRP along with TBS, aperiodic tasks must be assigned a suitable preemption level. Buttazzo et al, propose:

$$\pi_k = \frac{U_s}{C_k}$$

for each aperiodic task J_k. We can still use formula 2.5 ranging over the whole task set, to calculate the blocking using $D_j = \frac{C_j}{U_s}$ as deadline of aperiodic tasks.

The following theorem ensures schedulability for an hybrid set of tasks:

Theorem 2.7 (Lipari and Buttazzo,99) *Let T^P be a set of n periodic tasks ordered by decreasing preemption level ($\pi_i \geq \pi_j$ iff $i < j$) and let T^A be a set of aperiodic tasks scheduled by* TBS *with utilization U_s. Then, set T^P is schedulable by* EDF+SRP+TBS *if*

$$\sum_{i}^{n} \frac{E_i}{D_i} + U_s \leq 1 \tag{2.7}$$

$$\forall i, 1 \leq i \leq n, \forall L, D_i \leq L < D_n \tag{2.8}$$

$$L \geq \sum_{j=1}^{i} \lfloor \frac{L}{P_j} \rfloor E_j + max\{0, B_i - 1\} + LU_s \tag{2.9}$$

Example 2.11 *Consider two periodic tasks $T_1 = (2,8)$ and $T_2 = (3,12)$ which interact with two aperiodic jobs J_1 and J_2, both having execution time 2 and release times $r_1 = 0$ and $r_2 = 1$, respectively. $U_s \leq \frac{2}{8} + \frac{3}{12}$, $U_s = \frac{1}{2}$. $\pi_{J_1} = \pi_{J_2} = \frac{\frac{1}{2}}{2} = \frac{1}{4}$; T_1 and J_2 share the same resource during all their execution but J_2 has a higher preemption level. J_1 is served first in virtue of FIFO for aperiodic tasks and J_2 is served before T_1 even if both have the same preemption level, but we enhance responsiveness. Figure 2.8 shows the scheduling.*

2.6 Event Triggered Tasks

Up to now, we have described RTS as a collection of tasks, periodic and aperiodic, which are triggered by external events; implicitly for periodic tasks we assume the "period" as the *event* that makes a task (better said, a new instance of task) be released and enter the system. For aperiodic task, we are only interested in its arrival and in its scheduling taking into account other tasks already active.

We consider now RTS in which a task is triggered by *various* events in their environment. A task might be triggered as a consequence of another task completion or by various events in the environment.

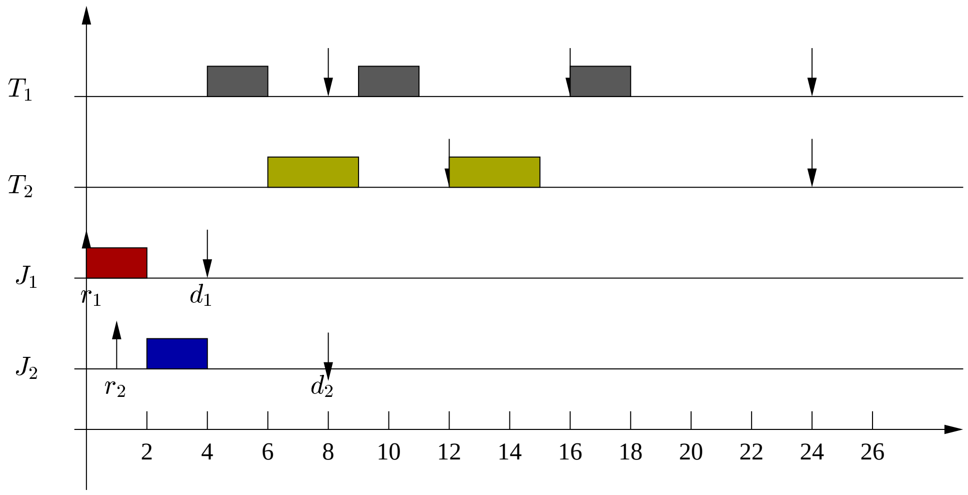


Figure 2.8: Sharing resources in an hybrid set

We will distinguish *internal* and *external* events; the former are related to the system itself and more precisely to the set of active tasks in the processor; the latter are related to the external environment, that is to the reaction of some procedures not included in the systems (for instance, sensors, measures instruments, human action, and so on).

Balarin et al, [13] have proposed an algorithm for schedule validation under a scene of event triggered tasks, ETT. We will describe their method as it sets up a new model for reactive RTS.

2.6.1 A Model for ETT

Intuitively an event triggered system is modelled as an execution graph, where some tasks are *enabled* by others or by some external events; feasibility of such a system is seen as *all tasks completing before a new occurrence of the event that triggers it re-appears in the system.* We say that a system is correct if certain critical events are never "dropped" or missed.

Formally, a system for ETT is a 6-uple (T, e, U, m, E, C) where:

- $T = \{1, 2, \ldots, n\}$ is a set of *internal task identifiers*, where identifiers also indicate tasks priority, the larger the identifier, the higher the priority. We note by π_i the priority of a task i.
- $e : T \rightarrow \Re^+$ which assigns to each internal task its (worst) execution time.
- U, such that $U \cap T = \emptyset$ is a set of unique *external task identifiers*, representing the tasks generated by external events of the environment.
- $m : U \rightarrow \Re^+$ which assigns to every external task the *minimum time between two occurrences of the event that triggers it.*
- $E \subseteq (T \cup U) \times T$ is a set of *events*; a pair (i, j) indicates that a task i (external or internal) enables the internal task j; if i is external, we say (i, j) is an external event, otherwise (i, j) is an internal event. Nodes $T \cup U$ and edges in E constitute the *system graph* of our application.

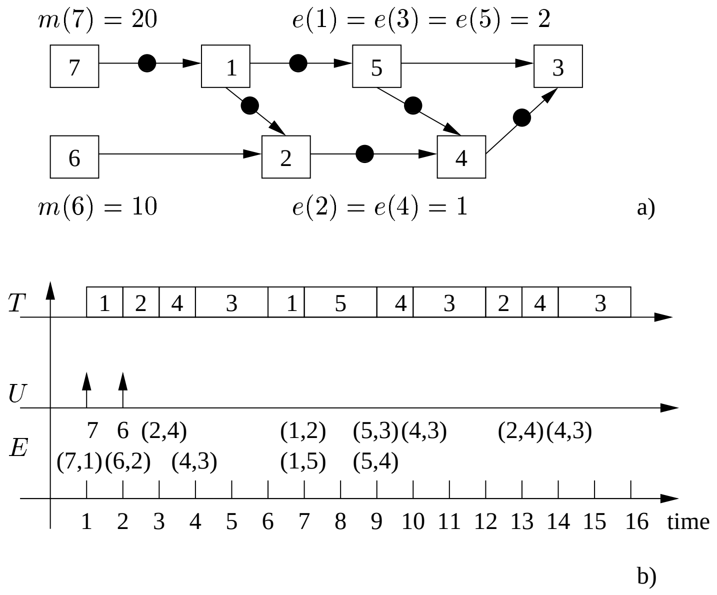


Figure 2.9: An example of ETT

- $C \subseteq E$ is a set of *critical events*

Example 2.12 *We show in figure 2.9a) a system with 7 tasks; tasks 6 and 7 are external, critical events are marked by dots and processing is not all inclusive, that is an internal task is triggered by one event. For instance, task 2 must start after receiving information from task 1 but need not wait for information from task 6 (event (6,2) is not critical and might be dropped).*

An *execution* of a system is a timed sequence of events that satisfies the following:

- An external task i can execute at any time, respecting the minimum delay $m(i)$ between two executions.
- after i has finished its execution, all tasks j such that $(i,j) \in E$ are enabled, and task i is disabled.
- If a task i is enabled at time t_1, then it will finish its execution or become *disabled* at time t_2 such that in interval $[t_1, t_2]$ the amount of time where i had the highest priority is $e(i)$.

An event (i,j) is *dropped* if after i becomes disabled, task i is executed again before task j is executed. An execution is *correct* if no critical events are dropped in it. A system is correct if all of its executions are. We show an execution of our example in figure 2.9b).

2.6.2 Validation of the Model

If we want to guarantee correctness, we need to show that no critical event is dropped in any execution; a sufficient condition for that is to ensure that for every critical event (i,j), the minimum time between two execution of i is larger than the maximum time between i and j. In Balarin's model, they propose a version where:

- Only external events can be dropped; the minimum time between two executions for these events is determined by a system description.
- Conservative estimation of the maximum time between execution of i and j.

The first proposition is quite simple:

Proposition 2.1 *If $i < j$ then (i, j) cannot be dropped.*

In fact, remember that in their model, $i < j$ implies $\pi_i < \pi_j$ and if i triggers j, this task has higher priority and can never be dropped by the arrival of a new instance of i.

Balarin et al. have settled a condition for an event (i, j) not to be dropped; it is based on the notion of an *exclusive frontier* for each internal task i.

Definition 2.6 (Exclusive Neighborhood) *Let (F, N) be a pair of disjoint subsets of tasks; (F, N) is an* exclusive neighborhood *for some internal task i if F and N satisfy the following conditions:*

C1 $i \in F \cup N$

C2 $\forall j, k : ((k \in N) \wedge ((j, k) \in E)) \rightarrow (j \in F \cup N)$

C3 $(\forall k \in F \cup N - i)\exists_1 j \in N : (k, j) \in E$ and i has no successors in N.

C4 $k < j$ for every $k \in F$ and every $j \in N$.

F is the frontier and N is the interior of an exclusive neighborhood, which gives the graph obtained by traversing backwards from i and stopping at the frontier nodes. For example, task 4 has an exclusive neighborhood with $F = \{1, 2\}$ and $N = \{4, 5\}$.

Under this definition, the following theorem holds:

Theorem 2.8 *If $(i, j) \in E$ and (F, N) is an exclusive neighborhood for task i, such that: $k < j \ \forall k \in F$, then event (i, j) cannot be dropped.*

which gives a very simple policy to assign priorities to tasks, based on propositon 2.1, which is in fact a corollary of this theorem.

On the other hand, we can verify if a critical event (i, j) can eventually be dropped; it suffices to perform a backward search of a system graph starting from i. The search finishes when we reach a task with priority less than j. If at any time some task is reached for the second time (violating C3) or an external task is reached (violating C4), the search finishes with failure (but results are inconclusive). On the contrary if no more unexplored nodes with priority larger than j are found, then we satisfy the theorem and the event cannot be dropped.

Finally the authors also propose a methodology to analyse the possibility of an external event be dropped, simplified in [12]. The problem is quite simple to formulate, but not easy to solve.

Basically, to know if an external event (i, j) can be dropped, we need to check whether the execution of j can be delayed for more than $m(i)$ units of time. In order to do so, they calculate an interval, called π_j-busy interval, where the processor is always servicing tasks with priorities higher than π_j. The first step in computing such a bound is to compute *partial loads*, noted $\delta(i, p)$, as the continuous load at priority p or higher caused by an execution of task i. At the beginning of a p-busy interval some task with priority lower than p, say k, may be executing and eventually at completion, k might enable some tasks of priority p or higher. The total workload generated by such a task is bounded by $\mathbf{max}\{\delta(k,p) \mid k \in T, \pi_k < p\}$.

As new tasks can be triggered as the consequence of external events, we consider that in a p-busy interval of length Δ, there can be at most $\lceil \frac{\Delta}{m(u)} \rceil$ executions of an external task u generating a workload of $\delta(u,p)$ at priority p or higher, hence we have:

$$\Delta \leq \mathbf{max}\{\delta(k,p) \mid k \in T, \pi_k < p\} + \sum_{u \in U} \lceil \frac{\Delta}{m(u)} \rceil \delta(u,p)$$

which can be solved by iteration; if $p = \pi_j$ and $\Delta < m(i)$, then (i,j) cannot be dropped.

2.7 Tasks with Complex Constraints

In this section, we present some ideas to attack the problem of scheduling when tasks must be analysed using complex constraints. We borrow from [30] the term *complex contraints* which means that a set of tasks is characterized not only by simple constraints such as period, release time and deadline but also by some other constraints which cannot be embedded in traditional scheduling.

Within these complex contraints, we can cite:

- *Precedence constraints*: such that a task is triggered by another task or the distribution of tasks in many processors which requires some internode communication.
- *Jitter*: even if a task must finish before its deadline, the evolution of a task may be different from instance to instance. The maximum time variation (relative to the release time) in the occurrence of a particular event in two consecutive instances of a task defines the jitter for that event. For example, the *start time jitter* of a task is the maximum time variation between the relative start times of any two consecutive jobs; similarly we can define the *response time jitter* as the maximum difference between the response times of any consecutive jobs, that is the maximum delay for an instance of a task, [19].
- *Non periodic execution*: where some instances of a tasks might be separated by non constant length intervals (this cannot be handled under EDF).
- *Semantic constraints*: tasks are characterized by parameters such as performance or relialiblity; for instance: allocate a task to a particular processor.

We will briefly describe the method proposed by [30] in order to handle tasks with complex constraints. The method begins by treating periodic tasks, which are reduced offline to create scheduling tables, [27]; it allocates tasks to nodes and resolves complex constraints by constructing sequences of task executions. Each task in a sequence is limited by either sending or receiving internode messages, predecessor or successor within the sequence. The final result is a set of independent tasks on single nodes with start-times and deadlines. These tasks can be scheduled according to traditional EDF method but we have to take into account the eventual arrival of aperiodic tasks which can violate the complex constraint construction.

Isovic et al. propose an extension of EDF, called *two level* EDF, [23]. There is a “normal level” to schedule tasks according to EDF but a “priority level” to an offline task when it needs to start at latest, similar to the basic idea of slack stealing for fixed priority scheduling, [57]. We need to know the amount and location of resources available after offline tasks are guaranteed schedulability.

For each node, we have a set of tasks, with start-times and deadlines, tasks are ordered by increasing deadlines and the schedule is divided into a set of disjoint execution intervals. For each instance j of offline task T_i we define a window $w(T_i^j)$. We have:

- $\mathrm{est}(T_i^j)$ which is expressed in the off-line schedule as the earliest start time, provided by the task constraints.
- $f(T_i^j)$ the scheduled finishing time according to the off-line schedule and
- $\mathrm{start}(T_i^j)$ the scheduled start time of instance j is the starting time of T_i^j according to the off-line schedule.

Each window $w(T_i^j) = [\mathrm{est}(T_i^j), dl(T_i^j)]$, where $dl(T_i^j)$ is the absolute deadline of instance j of task i.

We define *spare capacities* to represent the amount of available resources for these intervals. Each task deadline defines the end of an interval I_i. The start is defined as the maximum of the end of the previous interval or the earliest start time of the task. The end of the previous interval may be later than the earliest start time, or earlier, thus it is possible for a task to execute outside its interval, earlier than the interval start but never before its earliest start time.

The spare capacities of an interval I_i are calculated as:

$$sc(I_i) = |I_i| - \sum_{T \in I_i} E_T - \min(sc(I_{i+1}), 0)$$

since a task may execute prior to its interval, we have to decrease the spare capacities lent to subsequent intervals.

Runtime scheduling is performed locally for each node. If the spare capacities of the current interval are greater than 0, then EDF is applied on the set of ready tasks, -normal level. If no spare capacities are available, it means that a task has to be executed inmediately (since we have already guaranteed schedulability).

After each scheduling decision, the sc for an interval is updated. If a periodic task assigned to an interval I_c executes, no changes are need, but if a task T assigned to a later interval I_j, $j > c$ executes, the spare capacity of I_j is increased and that of I_c is decresead. We will show that current spare capacity is reduced by aperiodic tasks or idle execution.

When an aperiodic task J_i arrives to the system at time t_i we perform an acceptance test based on other previously arrived aperiodic task waiting for execution; if this set is called G, we should test if $G \cup J_i$ can be scheduled, considering offline tasks. If so, we can add J_i to G.

The finishing time of J_i, f_i, with execution C_i can be calculated with respect to J_{i-1}; with no offline tasks, $f_i = f_{i-1} + C_i$ represents the finishing time for J_i but we should extend the formula reflecting the amount of resources reserved for offline tasks:

$$f_i = C_i + \begin{cases} t + R[t, f_1] & i = 1 \\ f_{i-1} + R[f_{i-1}, f_i] & i > 1 \end{cases}$$

where $R[t_1, t_2]$ stands for the amount of resources reserved for the exectuion of offline tasks from time t_1 to t_2. We calculate this term by means of spare capacities:

$$R[t_1, t_2] = (t_2 - t_1) - \sum_{I_c \in (t_1, t_2)} \max(sc(I_c), 0)$$

As f_i appears on both sides of the equation, the authors propose an algorithm for acceptance of a new aperiodic task A_i in $O(n)$, where n is the number of aperiodic tasks in G not already completed.

In [23], Dobrin, Ozdemir and Fohler propose an algorithm for fixed priority assignment in the context of off-line tasks. For off-line tasks we assign priorities based on starting points; as the system evolves, it cannot always be possible to keep the same priority for different instances of the same task, so new 'ficticiuos' tasks are created.

Chapter 3

Inspiring Ideas

Résumé

Une première idée a été l'ordonnancement de programmes Java Temps Réel, pour répondre aux questions "est-ce qu'on peut modéliser un programme Java selon certains points d'observation?" et "est-ce qu'on peut trouver, à partir de ce modèle, un programm Java Temps Réel Ordonnancé?"

Pour resoudre ces deux problèmes on a commencé par l'ordonnancement à partir d'un certain model abstrait, [32].

Un programme Java en Temps Réel, est un ensemble S de threads H; chaque thread est indépendente à l'exécution mais elle se communique avec autres threads par les instructions de synchronisation.

Chaque thread H est divisée logiquement en *tâches*; chaque tâche d'une thread H peut être exécutée en parallèle ou entrelacée avec autres tâches d'autres threads. Les tâches d'une thread son ordonnancées d'une façon séquentielle.

Formellement, on dit qu'une thread H_j est composée par une séquence

$$\tau_1^j; \tau_2^j; \ldots; \tau_{n_j}^j$$

de tâches. Les ";" separent les differentes tâches à l'interieur d'une thread.

Chaque thread H_j peut être *périodique*, i.e. elle arrive dans certains intervalle de temps définis statiquement, T_j. A chaque tâche τ_i^j on va associer une valeur C_i correspondant au temps d'exécution. Finallement, à chaque thread H_j on peut associer une écheance, D_j correspondante au temps maximal de finition de la thread. C'est le cadre classique de STR, [54], [37], [36].

Dans le cadre de notre modèle, un *programme* est une séquence de tâches de differentes threads et un *programme ordonnancé* est une séquence de tâches telle que à l'exécution elle respecte les contraintes temporelles.

3.1 Introduction

Embedded systems play an increasingly important role in daily life. The strong increasing penetration of embedded systems in products and services creates huge opportunities for all kinds of enterprises and institutions, [3]. It concerns enterprises and institutions in such diverse areas as agriculture, health care, environment, road construction, security, mechanics, shipbuilding, medical appliances, language products, consumer electronics, etc. Real-time embedded systems interact continuously with the environment and have constraints on the speed with which they react to environment stimuli. Examples are power-train controllers for vehicles, embedded controllers for aircrafts, health monitoring systems and industrial plant controllers. Timing constraints introduce difficulties that make the design of embedded systems particularly challenging.

Hard RT embedded systems have tight timing constraints, i.e., they are difficult to achieve and they must not be violated, with respect to the capability of the hardware platforms used. Hard RT constraints challenge the way in which software is designed at its roots. Standard software development practices do not deal with physical properties of the system as a paradigm, so we need some new models which add non-functional aspects to the logic of the problems.

As embedded systems are growing, it stands out that a development language for these systems must be a popular programming language, which includes interesting features for real time environments. Java is a language which really covers many of the needs of real time programming, to the extent that today we can talk of *Real Time Java*, [15], RT-Java, and even scientific meetings concerning Java and Embedded Systems.

Java is a language which provides some basic components such as methods, grouped in classes and objects belonging to a class; it provides concurrency through the special class *Thread* where different processes can coordinate, wait and resume their executions; many of these needs are imperative in RTS. Another important feature of Java is its ortogonality, that is, almost everything is reduced to the concept of object.

Real Time Java deals easily with aspects such as scheduling, memory management, synchronization, asynchronous event handling and physical memory access, in some way platform-independent and hence applications are portable and the developement may be distributed. Java technology is already used in a variety of embedded applications, such as cellular phones and mobility.

Some of the advantages of the Java technology are:

- Portability. Platform independence enables code reuse across processors and product lines, allowing device manufacturers to deploy the same applications to a range of target devices and hence lower costs.
- Rapid application development. The Java programming language offers more flexibility during the development phase, since it can begin on a variety of available desktop environments, well before the targeted deployment hardware is available.
- Connectivity. The Java programming language provides a built-in framework for secure networking.
- Reliability. Embedded devices require high reliability. The simplicity of the Java programming language, -with its absence of pointers and its automatic garbage collection-, eliminates many bugs and the risk of memory leaks.

It stands out that Java is a language which fulfils many of the real time requirements over a firm language architecture; even more, Java is very popular and well known for the implementation community.

Java and Schedulability. Our first need was in principle to answer the question "is it possible to model a RT-Java Program in order to synthesize a scheduled program which ensures all temporal

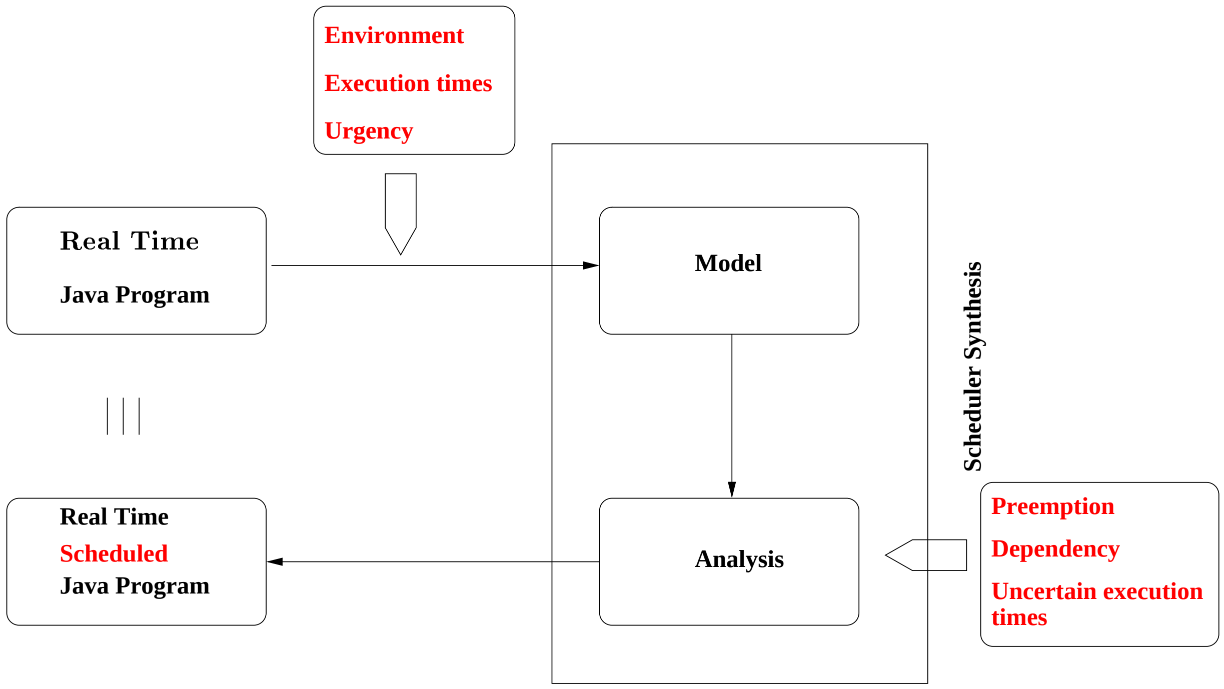


Figure 3.1: Construction of a RT-Java Scheduled Program

contraints?". Our objective can be resumed in figure 3.1, where a Java program suffers a process of analysis in order to construct, *synthetise*, the scheduled program.

So, we need to model a Java Program or a RT-Java Program in order to perform scheduling operations. We are particularly interested in analysing a program to say whether it is schedulable or not. Scheduling properties to be respected are deadlines, execution times, synchronization points and shared resources. The final objective is to find a possible sequence of execution which can guarantee all the properties mentioned above.

The result of the analysis should be a scheduled Java program with some scheduling policy embedded in the Java language through its RT platform. Java provides some means to model synchronization among processes, through two primitives *wait* and *notify* and mutual exclusion through the attribute *synchronized* over an objet. Java performs coordination by blocking an object. So, to be independent of this special semantics, we propose to differentiate clearly these two aspects:

1. Synchronization or coordination among threads, that is communication in a producer/consumer fashion is done through two primitives: **await** to signal waiting of a message and **emit** to signal sending of a message. Conceptually speaking it is as if there were no explicit locking of the object over which we wait.

2. Mutual exclusion, that is an object cannot be accessed by more than one thread at a time, in order to assure correctness. This is done through the (Java) attribute **synchronized** over the object which must be preserved.

```
class PeriodicTh extends PeriodicRTThread
{ long p ;
  ThreadBody b ;
  PeriodicTh(long p, ThreadBody b)
  {   this.p = p ;
      this.b = b ;
  }
  public void run()
  { long t ;
    Clock c = new Clock() ;
    while(true)
    { t = c.getTime() ;
      b.exec() ;
      waitforperiod(p + t - c.getTime());
    }
  }
}

interface ThreadBody
{    public void exec() ;
}

class Thread1_body implements ThreadBody
{ Event a, b ;
  Thread1_body (Event a, b)
  { this.a = a ;
    this.b = b ;
  }
  public void exec()
  { t7 ;
    t1 ;
    a.emit;
    t5 ;
    b.emit;

  }
}
```

```
class Thread2_body implements ThreadBody
{ Event a, b ;

  public void exec()
  { t6;
    a.await;
    t2;
    b.await;
    t4;
    t3;
  }
}

class Example
{ public static void main(String argv[])
  { Event a = new Event() ;
    Event b = new Event() ;
    Thread1_body th1_body = new Thread1_body(a,b) ;
    Thread2_body th2_body = new Thread2_body(a,b) ;
    PeriodicTh thread1 = new PeriodicTh(10, th1_body) ;
    PeriodicTh thread2 = new PeriodicTh(20, th2_body) ;
  }
}

class Event
{ public void emit()
  { synchronized(this) {this.notify}
  }

  public void await()
  { synchronized(this) {this.wait}
  }
}
```

Figure 3.2: Two Threads

We present in figure 3.2 an example of a Java-like program where we have modified some of its primitives.

We also need to coordinate the environment and the application through the execution platform. The environment is represented by a series of events which may be triggered by time passing or by a control device; they must be taken into account by the application in some predefined delay, but the application response depends greatly on the speed of the execution architecture.

3.2 Model of a RT-Java Program

We model a Java Program as a set T of Threads, each thread is independent in its execution but it communicates to other threads through *await* and *emit* instructions to cooperate in the execution of a task, and *synchronized blocks* to coordinate access to critical sections of common resources in a mutually exclusive manner.

Threads and Tasks. Each thread H is *logically divided* into blocks of instructions, which we call *tasks*; certain tasks can be executed in parallel or in an interleaved way with other tasks of other threads, but tasks within the same thread H are *sequentially ordered.*

Formally, we can say that a thread H_i is composed by a sequence

$$\tau_1^i; \tau_2^i; \ldots; \tau_{n_i}^i$$

of tasks (note the ";" separating the tasks).

This model can be obtained by application of some techniques such as [32, 28] where some observation points are considered to "cut up" the code. We are particularly interested in synchronization among threads through the operations *await* and *emit* and use of shared resources.

Each thread H_i can be *periodic*, that is, it arrives at regular intervals of time, defined statically. We note P_i the period for thread H_i. Each task τ_k^i has a (worst case) *execution time*, E_k^i which is also static and derives from some off line analysis. Finally, we associate a *deadline* D_i to each thread H_i. This is the classical approach for RTS, [37, 36, 54], which we developped in chapter 2.

In our model, a *program* is a sequence of tasks from different threads and a *scheduled program* is a sequence of tasks which in execution will respect the timing constraints (deadlocks, execution times, periods).

Tasks and Resources. Tasks in H may access some shared resources, that is, shared data whose access must be protected by a protocol to guarantee that at most one and only one modifier is present at any time. As we have seen, before accessing a shared resource, r_i, a *locking* operation over r_i is demanded to the data manager who keeps a register of all resources and their states (free or busy); such operation may be *granted* if the resource is free or *denied* if it is busy, in this case, the demander waits for permission.

Once a task has finished with r_i it releases it to the system by an *unlock* operation, u_i, which is always successful. We demand an "ordered" usage of lock and unlock operations, that is the last locked resource is the first to be unlocked, following a stack logic.

In Java we recognize the lock and unlock operation by the structure:

```
...
synchronized(r1)
{ ...
  ...
}
...
```

where the block between "{" and "}" is the critical section for r_1 and *synchronized* is a modifier of the block indicating that *before* accessing this code, we must obtain a lock over the object, (r_1 in our case), equivalent to a *lock* operation. After exiting this protected code, the lock over r_1 is released.

For a set T of threads, we define the set $\mathbb{R}$ as the universal set of all shared resources used by tasks in T and to each τ_k^i, we associate a set $R(\tau_k^i) \subseteq \mathbb{R}$ of the resources it needs.

We are now ready to give the following definition:

Definition 3.1 *A* **scheduled program** *is a sequence of tasks of different threads which in execution respects the timing constraints (absence of deadlocks, execution times and periods) and mutual exclusion for shared resources.*

Relationships Among Tasks. If we consider two tasks τ_k^i and τ_l^j we can establish one of the following relations:

1. τ_k^i, τ_l^j are **independent**, $i \neq j$ and they can be executed in any order, that is, they belong to different threads, they do not share resources and they do not coordinate.

2. τ_k^i, τ_l^j belong to the same thread H_i, $i = j$, and will be executed according to the internal logic of H_i: τ_k^i is executed before τ_l^j, if $k < l$. We denote $\tau_k^i; \tau_l^i$ the **immediate precedence relation** (in fact, $l = k + 1$) of two tasks from thread H_i and $\tau_k^i \rightarrow \tau_l^j$, the **precedence relation** in the sequence of the decomposed thread H_i, i.e., the transitive closure of the sequence relation, ";".

3. τ_k^i, τ_l^j belong to different threads and communicate through a **await/emit** relation. In this case we can say that τ_k^i **notifies** τ_l^j, denoted $\tau_k^i \rightsquigarrow \tau_l^j$. The $\rightsquigarrow$ relation, expresses a waiting state for τ_l^j until the emit arrives, that is we can see τ_k^i as a producer and τ_l^j as a consumer and the emit as the fact that a product is ready. On the other hand, τ_l^j must be in a waiting state to "hear" a notify. To each thread H_i, we associate the set N_i of *notifiers* that is:
$$\mathcal{N}_i = \{\tau_k^i | \tau_k^i \rightsquigarrow \tau_l^j, i \neq j\}$$

4. τ_k^i, τ_l^j use a common resource r, then $\tau_k^i \overset{r}{\leftrightarrow} \tau_l^j$ if $r \in R(\tau_k^i) \wedge r \in R(\tau_l^j)$.

It should be clear that both the precedence and the wait relations impose a hierarchical relation between two tasks, but the await/emit relation imposes a coordination with another task, while the precedence relation is simply a way to express that a task will be thrown after the completion of its preceding in the sequence.

Precedence can be established statically and it is always "successful" in execution time, while await/emit relation may fail if the waiting task is not ready to hear a notify; in our model, the scheduler must assure this procedure in order to guarantee success of the operation.

In this hierarchy we distinguish some special tasks:

- Task τ_a^H is the *starting* of a thread H if $\forall k, \tau_a^H \rightarrow \tau_k^H$
- Task τ_z^H is the *last* of a thread H if $\forall k, \tau_k^H \rightarrow \tau_z^H$
- Finally, task τ_k^i is *autonomous* if it does not wait for another task, that is if $\neg\exists l, j\ \tau_l^j \rightsquigarrow \tau_k^i$.

3.2.1 Structural Model

We model a program as a graph, where the set of nodes corresponds to tasks and the set of arcs corresponds to precedence and await/emit relations. We describe our model through an example.

Example 3.1 *Figure 3.3 shows the model of the program in figure 3.2.*

We can observe two threads H_1 and H_2 composed by the sequence
$$H_1 = [\tau_7; \tau_1; \tau_5]$$
and
$$H_2 = [\tau_6; \tau_2; \tau_4; \tau_3]$$

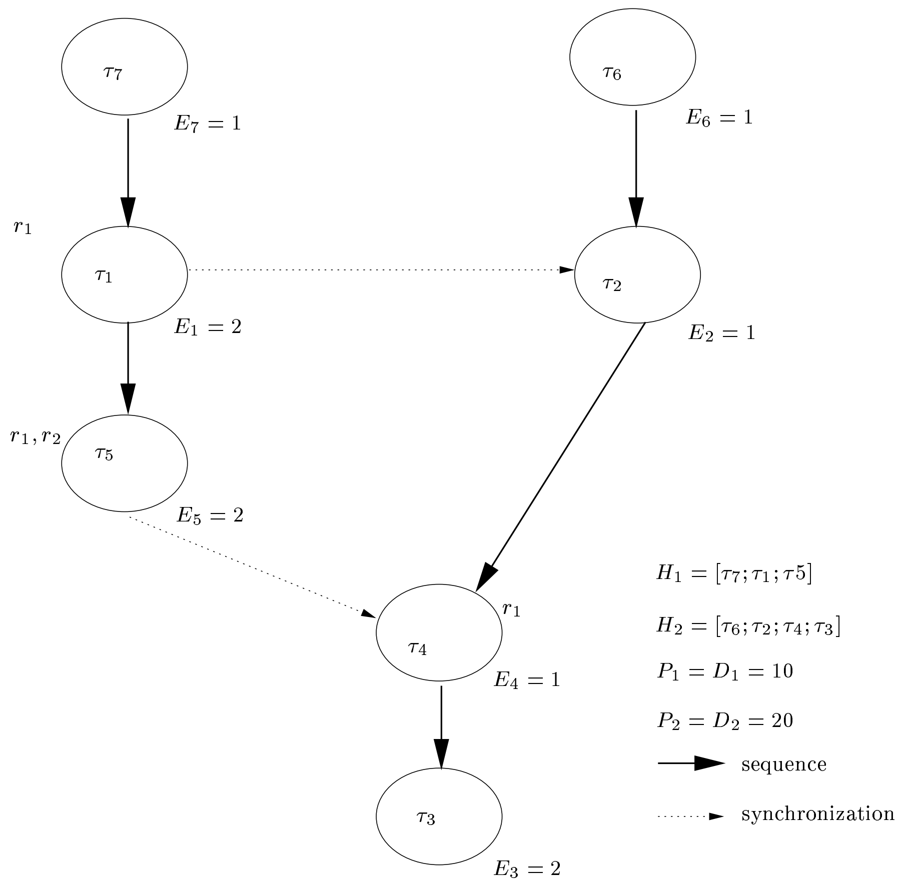


Figure 3.3: Two Threads

respectively[1]*; we can also see two synchronization points as* $\tau_1 \rightsquigarrow \tau_2$ *and* $\tau_5 \rightsquigarrow \tau_4$*, shown as dotted lines; worst execution time for each of the tasks is indicated beneath each task.*

$\mathbb{R} = \{r_1, r_2\}$*, task* τ_1 *uses a resource* r_1 *and* τ_5 *uses both* r_1 *and* r_2*; task* τ_4 *uses* r_1 *and* $\tau_1 \overset{r_1}{\leftrightarrow} \tau_4$ *among others.*

The model of the program shows a partial order among tasks; those belonging to the same thread are *totally ordered*, by the sequence relation; those tasks tied by a $\rightsquigarrow$ relation are also totally ordered and finally some tasks are not ordered.

3.2.2 Behavioral Model

Task behaviour can be described through a classical state model shown in figure 3.4, which is self-explanatory. Anyway let us note that the execution platform has three queues: ready (RQ), waiting (WQ) and sleeping (SQ), associated to the respective states. Each task can be in one of the following states:

[1]we will skip the superindex indicating the thread if no confusion results

- Idle: task is not active.
- Ready: task is in RQ and can be chosen by the scheduler to begin execution. It needs no *emit* operation but may need or even have some shared resources.
- Waiting: task is in WQ, waiting for an emit; its execution is blocked until the emit arrives.
- Executing: task is running.
- Sleeping: task is in SQ because it was preempted by a higher priority task. Later it will resume its execution (it is not blocked).

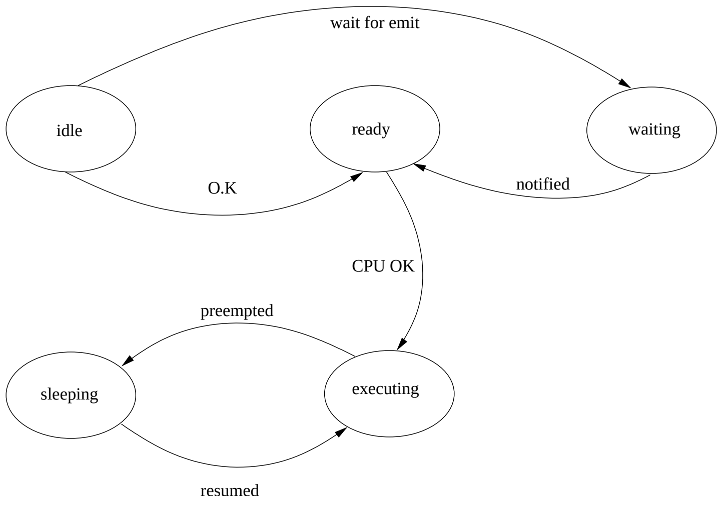


Figure 3.4: State Model

We define the following rules to manage the queues over the execution:

Ready Rule

$$\frac{\tau_k^i \uparrow \wedge \neg \exists \tau_l^j, i \neq j, \tau_l^j \rightsquigarrow \tau_k^i}{\text{RQ} \rightarrow \text{RQ} \oplus \tau_k^i}$$

Waiting Rule

$$\frac{\tau_k^i \uparrow \wedge \exists \tau_l^j, i \neq j, \tau_l^j \rightsquigarrow \tau_k^i}{\text{WQ} \rightarrow \text{WQ} \oplus \tau_k^i}$$

Migration Rule

$$\frac{\tau_k^i \in \text{WQ} \wedge [\exists \tau_l^j, \tau_l^j \downarrow, i \neq j \wedge \tau_l^j \rightsquigarrow \tau_k^i]}{\text{RQ} \rightarrow \text{RQ} \oplus \tau_k^i \wedge \text{WQ} \rightarrow \text{WQ} \ominus \tau_k^i}$$

Preemption Rule

$$\frac{\text{exec}(\tau_k^i) \wedge [\exists \tau_l^j, \tau_l^j \in \text{RQ}, i \neq j, \pi_k^i < \pi_l^j \wedge \neg \text{locked}(R(\tau_l^j))]}{\text{SQ} \rightarrow \text{SQ} \oplus \tau_k^i}$$

Execution Rule

$$\frac{\tau_k^i \in \mathrm{RQ} \wedge \pi_k^i > \mathrm{highest(RQ)} \wedge \pi_k^i > \mathrm{highest(SQ)} \wedge \neg\mathrm{locked}(R(\tau_k^i))}{\mathrm{RQ} \rightarrow \mathrm{RQ} \ominus \tau_k^i \wedge \mathrm{exec}(\tau_k^i)}$$

Resuming Rule

$$\frac{\tau_k^i \in \mathrm{SQ} \wedge \pi_k^i > \mathrm{highest(RQ)} \wedge \pi_k^i > \mathrm{highest(SQ)} \wedge \neg\mathrm{locked}(R(\tau_k^i))}{\mathrm{SQ} \rightarrow \mathrm{SQ} \ominus \tau_k^i \wedge \mathrm{exec}(\tau_k^i)}$$

- The $\oplus$ and $\ominus$ represent the queueing and dequeueing operations, respectively;
- $\tau\uparrow$ and $\tau\downarrow$ represent the arrival and completion of task τ;
- Predicate:
  - exec(τ) indicates that τ is executing,
  - locked($R(\tau)$) the fact that τ is locked by one or more resources and
  - highest(Q) gives the maximum priority in queue Q.
- π_k^i is the priority of τ_k^i; next section clarifies priority assignment.

Remark 1 Note that a task that does not wait for an *emit* is in the RQ, with some priority; if it has the highest priority and all resources it needs, it executes. Preemption, based on priorities, is permitted.

Remark 2 If a task is in the WQ then it needs an *emit* from some other task; it can wait for an *emit* retaining a resource locked (and never released) by one of its ancestors but it cannot be waiting for an *emit and a new resource at the same time*, since the *await* operation prevents execution.

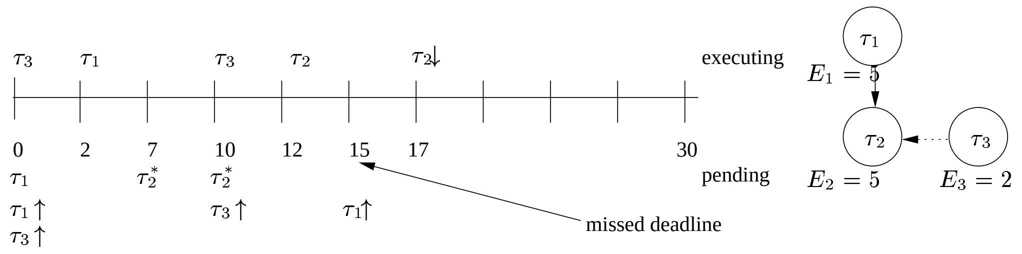
3.3 Schedulability without Shared Resources

A scheduling algorithm gives some order among tasks; in a static or dynamic manner, this order is based on some restrictions and relationships among tasks, which can lead the scheduler to some decisions. As already said this order is based on timing constraints since a task must respond within its deadline or it may cause a critical event to happen; in our model we need also to schedule the precedence and the await/emit relation. For the instant being, we are not considering shared resources.

We have defined a simple *fixed priority assignment algorithm*, which takes into account the precedence and await/emit relations:

Rule–I To each thread H_i we assign a priority Π_i based on some classical fixed scheduling policy, such as RMA; these policies take into account the period, P_i, or deadline, D_i, of threads. For instance, in the case RMA we can say that $\Pi_i > \Pi_j$ if $P_i < P_j$. This is the *base priority* for all tasks in H_i.

Rule–II If $\tau_k^i;\tau_{k+1}^i \wedge \tau_l^j \rightsquigarrow \tau_{k+1}^i \wedge (i \neq j) \Rightarrow \pi_k^i > \pi_l^j$. The second rule changes the priority to some tasks within a thread.



τ_3 τ_1 τ_3 τ_2 $\tau_2\downarrow$
executing
0 2 7 10 12 15 17 30
τ_1 τ_2^* τ_2^*
$\tau_1\uparrow$ $\tau_3\uparrow$ $\tau_1\uparrow$
$\tau_3\uparrow$
pending
missed deadline
τ_1
$E_1 = 5$
τ_2 τ_3
$E_2 = 5$ $E_3 = 2$

Figure 3.5: Counter example of priority assignment

If all tasks were independent, the first rule suffices to execute each thread autonomously and following an RMA analysis we could know of its schedulability (see chapter 2).

The second rule applies to the operation of emit. Remember that a task waiting for an emit is in the WQ (waiting rule) and will remain there until it "hears" such operation. On the other hand, an emit operation is always "successful": the notifier sends an emit and continues its execution (it is to the execution platform to manage this operation), but the waiter **must** be in a waiting state to listen the notify. This rule states that a waiting task, τ^i_{k+1}, will be triggered by its ascendent τ^i_k, and put into the WQ *before* the execution of τ^j_l from which it waits the emit, in order to be ready to "hear" it and be ready to execute (migration rule). Observe that the starting task τ^H_a of a thread never waits.

In conclusion:

$$\forall \tau^i_k, \tau^j_l, i \neq j, \pi^i_k > \pi^j_l \text{ iff} \begin{cases} \Pi_i > \Pi_j \wedge \tau^i_k \notin \mathcal{N}_i \\ \text{or} \\ \exists \tau^i_{k+1} \wedge \tau^j_l \rightsquigarrow \tau^i_{k+1} \end{cases}$$

which gives a partial set of constraints of the form $\bigwedge_{i\neq j} \pi^i_k \theta \pi^j_l$, and $\theta \in \{<,>\}$.

Example 3.2 *(See figure 3.5) Let us consider two threads $H_A = [\tau_1; \tau_2]$ of period 15, where both tasks have an execution time of 5 and a thread $H_B = [\tau_3]$ of period 10 and τ_3 executes in 2 units and then notifies τ_2; both threads arrive at $t = 0$. Considering periods as deadlines and fixing priorities using exclusively* RMA *gives $\pi_3 > \pi_1 = \pi_2$. Under this assignment, the execution shows a deadline missing.*

On the contrary, if we use our rules, we have that no relation can be stablished among τ_2 and τ_3 and $\pi_3 < \pi_1$ (since $\tau_1; \tau_2 \wedge \tau_3 \rightsquigarrow \tau_2$), the system becomes schedulable.

3.3.1 Model Analysis

To each task τ^i_k we can associate the following "times":

- Arrival time, α^i_k, denoting the time a task is put in the ready or waiting queue.
- Blocking time, β^i_k, denoting the time a task is retained in the ready or waiting queue.
- Sleeping time, σ^i_k, denoting the time a task spends in the sleeping queue, after being preempted.
- Finishing time, f^i_k, denoting the time a task complets its execution.

Definition 3.2 *The* **starting time** *of a thread H_i, α_i, is the arrival time of its starting task, i.e., $\alpha_i = \alpha_a^i$ and $\tau_a^i = starting(H_i)$*

Definition 3.3 *The* **finishing time** *of a thread H_i, F_i, is the finishing time of the last task in the thread, i.e., $F_i = f_z^i$ and $\tau_z^i = last(H_i)$*

Definition 3.4 (Schedulable Thread) *A thread H_i is* **schedulable** *if $F_i \leq D_i$, that is, its execution finishes before its deadline.*

Definition 3.5 (Schedulable System) *A set $T = \{H_1, H_2, \ldots, H_n\}$ of threads composing an application is* **schedulable** *if all threads H_i are schedulable, i.e,*

$$\forall H_i, 1 \leq i \leq n, F_i \leq D_i$$

As noted in chapter 2, [37], to verify schedulability it sufficies to analyse the *time-window* or interval $[0, \mathcal{H}]$, where $\mathcal{H}$ is the hyperperiod for all periodic threads, defined as the least common multiple of all periods. For each thread H_i we see its evolution within the interval and if at arrival of a new instance of its starting task, the finishing task corresponding to the precedent execution has already finished, the thread is schedulable. This idea motivates the following revisited definitions:

Definition 3.6 (Finishing Time Periodic Task) *The* **finishing time** *of a task τ_k^i of a periodic thread H_i in its j-period is calculated as*

$$f_k^{i,j} = \alpha_k^{i,j} + \beta_k^{i,j} + \sigma_k^{i,j} + E_k^i$$

Definition 3.7 (Finishing Time Periodic Thread) *The* **finishing time** *of a periodic thread H_i in the j-period, that is F_i^j, is the finishing time of its last task, in the j-period,*

$$F_i^j = f_z^{i,j}$$

where $\tau_z = last(H_i)$.

Definition 3.8 (Schedulable Periodic Thread) *A periodic thread H_i is* **schedulable** *if*

$$F_i^j \leq \alpha_i^j + P_i = j \times P_i$$

for all $j, 1 \leq j \leq \eta_i$, where j is a period, η_i is the number of periodic arrivals of H_i within $[0, \mathcal{H}]$

If a system respects the previous rule for all its threads, we have a schedulable application.

Definition 3.9 (Schedulable Periodic System) *An application system of periodic threads, $T = \{H_1, H_2, \ldots, H_n\}$ is* schedulable *if all threads H_i are schedulable:*

$$\forall i \; 1 \leq i \leq n, F_i^j \leq \alpha_i^j + P_i$$

where $1 \leq j \leq \eta_i$, j is a period, η_i is the number of periodic arrivals of H_i within $[0, \mathcal{H}]$

Resuming, we present an operational approach of our model.

1. Priorities are assigned off line according to rules 1 and 2.

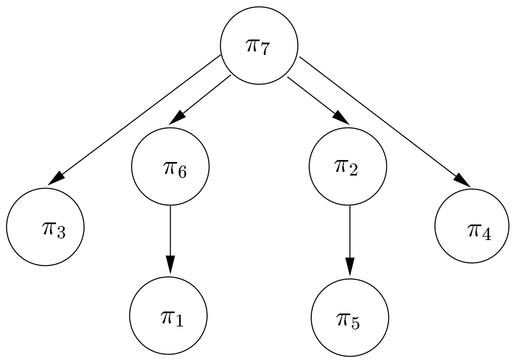


Figure 3.6: Partially Ordered Tasks

2. At time $t = 0$ starting tasks of active threads are in RQ.
3. The highest priority starting task of a thread H_i begins its execution.
4. When a task τ_k^i finishes, it may trigger another task τ_{k+1}^i in the sequence, which is put in the RQ (ready rule) or WQ (waiting rule).
5. When a task τ_k^i finishes it may **emit** to τ_l^j; accordingly to the migration rule, τ_l^j is awakened, if it is in the WQ and it is sent to RQ; otherwise the event is lost.
6. If at a moment $t = t'$ it arrives a task τ_l^j which has greater priority than that in execution, say τ_k^i, then τ_k^i is preempted.

3.3.2 Examples

Example 3.3 *Let us reconsider our example 3.1, without resources.*

According to the operational approach, we assign threads (and tasks within threads) priorities using our rules; applying this criteria to our example, gives:

1. $\Pi_1 > \Pi_2$ since $P_1 < P_2$
2. As $\tau_6; \tau_2$ and $\tau_1 \rightsquigarrow \tau_2$ then $\pi_6 > \pi_1$
3. As $\tau_2; \tau_4$ and $\tau_5 \rightsquigarrow \tau_4$ then $\pi_2 > \pi_5$

According to this mechanism, the rest of the tasks have the same base priority of their threads or, in other terms, they *inherit* the priority of their threads. We show in figure 3.6 the partial order obtained by application of our rules.

Remark Observe that certain tasks, such as τ_1 and τ_3, are not comparable; we can establish some priority order among them based on a fixed criteria. For instance, $\pi_1 > \pi_3$ if we consider that τ_1 belongs to a thread with higher priority than that of τ_3's. Tasks from a thread are naturally ordered by the precedence relationship.

Now let us put our example in operation, as should be done by the scheduler implementing our approach, supposing a starting time of t=0; the following table shows a possible result:

	Period 1					Period 2				
task	α_k^i	β_k^i	σ_k^i	$f_k^{i,1}$	d_k^1	α_k^i	β_k^i	σ_k^i	$f_k^{i,2}$	d_k^2
τ_7	0	0	0	1	10	10	0	0	11	20
τ_6	0	1	0	2	20					
τ_1	1	1	0	4	10	11	0	0	13	20
τ_2	2	2	0	5	20					
τ_5	4	1	0	7	10	13	0	0	15	20
τ_4	5	2	0	8	20					
τ_3	8	0	0	10	20					

We show in figure 3.7 the time line, where tasks in the upper part are those in execution and those in the lower part are in the ready or waiting queue.

1. At $t = 0$ the system is initiated, entering both τ_7 and τ_6 to the ready queue.
2. As τ_7 has greater priority, it is chosen to be executed and sent to the execution state.
3. As τ_7 finishes it triggers τ_1 which is also sent to the ready queue (τ_1 is autonomous). The scheduler chooses τ_6 (see point 2 priority assignment).
4. τ_6 is executed and it triggers τ_2, which is sent to the WQ (waiting rule).
5. The scheduler executes τ_1 (in fact, the only task in the ready queue).
6. When τ_1 finishes it triggers τ_5, which is sent to the ready queue; τ_1 awakes τ_2, which also goes to the RQ. Priorities analysed, the scheduler chooses τ_2 (see point 3 priority assignment).
7. When τ_2 finishes it triggers τ_4 which is sent to the WQ.
8. τ_5 executes and when it finishes, it awakes τ_4 which goes to the RQ. H_1 is finished at time $t = 7$.
9. τ_4 is executed and triggers τ_3.
10. At $t = 8$, τ_3 is executed and finishes at time $t = 10$; H_2 is finished.

The time analysis according to this schedule says that τ_5 finishes at time $t = 7$, ready to process another arrival of tasks of the next period of H_1; τ_3 finishes at time $t = 10$ ready to process a new instance of H_2. As both threads finish before their deadlines, with no pending tasks in queues, the system is schedulable in this first "round".

The second round for H_1 is a little simpler, as there are no tasks from H_2:

1. At $t = 10$, τ_7 arrives to the system and it is executed.
2. As τ_7 finishes, it triggers τ_1, which is executed; at completion it sends an emit to τ_2 which is lost and it triggers τ_5.
3. τ_5 is executed and analogously it notifies τ_4, event that is also lost.

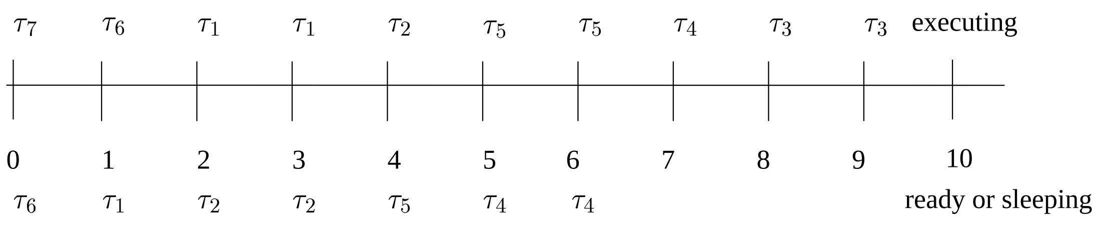

τ_7 τ_6 τ_1 τ_1 τ_2 τ_5 τ_5 τ_4 τ_3 τ_3 executing
0 1 2 3 4 5 6 7 8 9 10
τ_6 τ_1 τ_2 τ_2 τ_5 τ_4 τ_4 ready or sleeping

Figure 3.7: Time Line for ex. 3.1

The system is finished at time $t = 15$, remaining idle until $t = 20$, when a new set of periodic tasks from H_1 and H_2 will arrive, repeating the same pattern. The analysis of time in the interval $[0, \mathcal{H}]$, where $\mathcal{H}$ it is the hyperperiod of all the periodical threads, is sufficient to say that the system is schedulable (provided both threads start at the same time). Note that $F_1^1 = 7 < 10$ and $F_1^2 = 15 < 20$ and $F_2^1 = 10 < 20$.

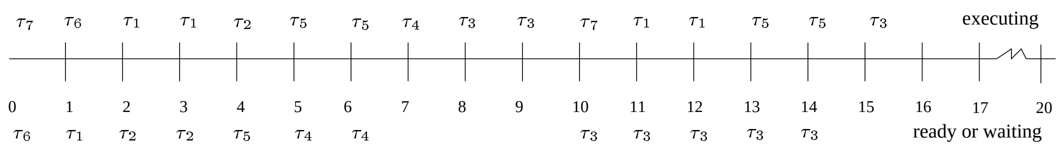
Example 3.4 *We will now modify our example, setting the execution time for τ_3 to 3, that is, $E_3 = 3$. The following table illustrates the reaction of our scheduler:*

	Period 1					Period 2				
task	α_k^i	β_k^i	σ_k^i	$f_k^{i,1}$	d_k^1	α_k^i	β_k^i	σ_k^i	$f_k^{i,2}$	d_k^2
τ_7	0	0	0	1	10	10	0	0	11	20
τ_6	0	1	0	2	20					
τ_1	1	1	0	4	10	11	0	0	13	20
τ_2	2	2	0	5	20					
τ_5	4	1	0	7	10	13	0	0	15	20
τ_4	5	2	0	8	20					
τ_3	8	0	5	16	20					

The procedure is exactly the same as before, except for the last point 10, where τ_3 is executing (see figure 3.8 for the time line):

1. τ_3 begins at $t = 8$, and it executes for 2 units, when τ_7 arrives for the next period. As τ_7 has greater priority than τ_3, the latter is preempted and sent to the SQ, (preemption rule).
2. τ_7 is executed until completion and triggers τ_1.
3. No priority relation is established among τ_1 and τ_3; if we consider a RMA criteria τ_1 has higher priority. Let us say that the scheduler chooses τ_1 based on this criteria, then τ_3 remains for 2 additional units in the SQ.
4. Once τ_1 finished, it triggers τ_5; τ_1's emit is lost.
5. τ_5 has greater priority than τ_3 (for the same reason as before); τ_3 remains for two more units of time in the SQ.
6. At τ_5 completion, τ_3 regains the processor and finishes at $t = 16$.

As $F_1^1 = 7 < 10$ and $F_1^2 = 15 < 20$, H_1 is schedulable and as $F_2^1 = 16 < 20$, H_2 is also schedulable; 20 is the lcm, so it sufficies to assure schedulability within the interval $[0, 20]$ to assure schedulability for the whole system.



τ_7 τ_6 τ_1 τ_1 τ_2 τ_5 τ_5 τ_4 τ_3 τ_3 τ_7 τ_1 τ_1 τ_5 τ_5 τ_3
executing
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 20
τ_6 τ_1 τ_2 τ_2 τ_5 τ_4 τ_4 τ_3 τ_3 τ_3 τ_3 τ_3
ready or waiting

Figure 3.8: Time Line for ex. 3.4

3.4 Sharing Resources

We will now consider the possibility of sharing resources among tasks; the gold rule is to prevent two or more task to access simultaneously the same resource, so our algorithm must impose a *mutual exclusion policy*.

As we consider a fixed set of tasks, that is, no eventual tasks can arrive during execution, we want some static analysis within the hyperperiod to decide if the system is schedulable and if so, assign priorities in order to guarantee timing constraints and mutual exclusion. Decisions taken by the scheduler are based on the states of each of the active tasks, *but* this analysis should be off line to minimize scheduler invasion during tasks execution.

Note that synchronization implies a certain order of execution among tasks, due to a some producer/consumer relation among them, while sharing resources implies a synchronization to respect the mutual exclusion rule but *no order* is implied.

Example 3.5 *Let us reconsider our example 3.1 of figure 3.3; figure 3.9 shows the corresponding Java code and the model generated by application of an abstraction algorithm. Note the "separation" from a waiting task and a demand of resource in τ_4^0 and τ_4, which is immaterial in our previous analysis since no resources are considered. Now, let us see how our assignment works in the presence of resources (the time line in figure 3.10 shows the evolution of tasks in time):*

1. τ_7 has the highest priority but $\pi_1 < \pi_6$ and $\pi_5 < \pi_2$ due to the await/emit relation.

2. τ_7 begins execution and τ_6 goes to the RQ.

3. τ_7 triggers τ_1 which goes to the RQ and

4. τ_6 is chosen to be executed; at completion it triggers τ_2 which goes to the WQ.

5. τ_1 is executed, setting the lock over r_1 and at its completion it emits to τ_2 and triggers τ_5, which goes to the RQ with r_1 retained.

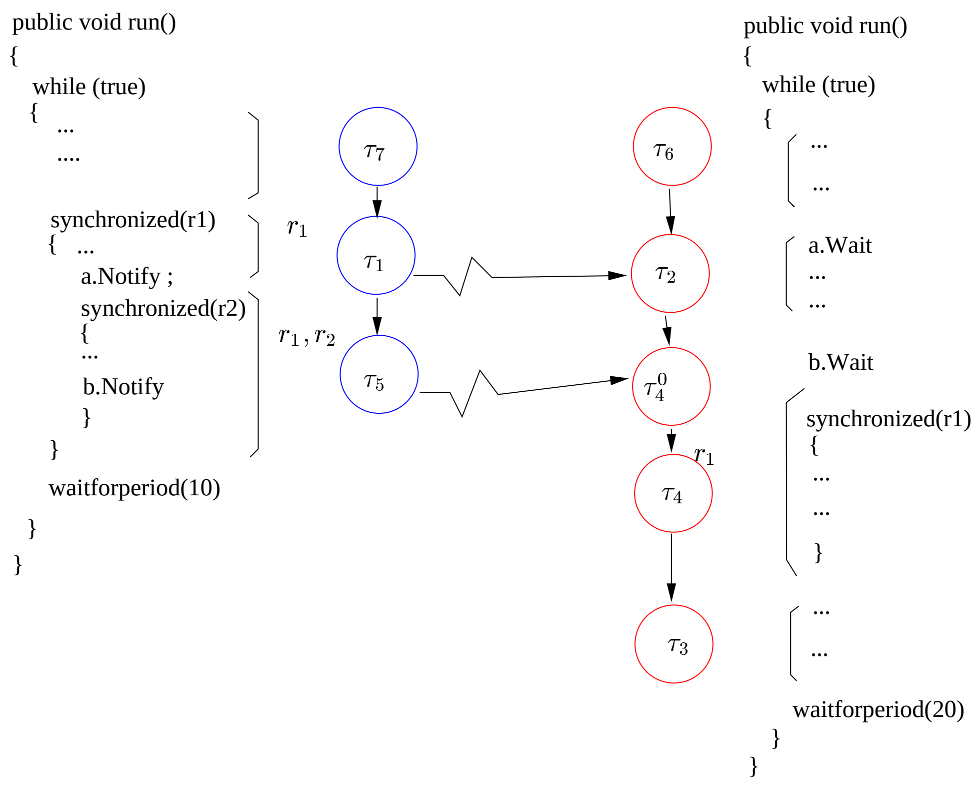
6. τ_2, awaken by τ_1 goes to the RQ and joins τ_5; $\pi_2 > \pi_5$, τ_2 is chosen to be executed, and at completion it triggers τ_4 which goes to the WQ.

7. τ_5 executes, releases its lock over r_1 and r_2, and notifies τ_4^0, which goes to the RQ "as" τ_4.

8. τ_4 executes, (over r_1), releases r_1 and triggers τ_3.

9. τ_3 executes and finishes at $t = 10$.

10. At $t = 10$ the next period of H_1 arrives and τ_7 is executed, triggering τ_1.



```
public void run()
{
   while (true)
  {   ...
      ....

      synchronized(r1)
      {   ...
          a.Notify ;
          synchronized(r2)
          {
          ...
          b.Notify
          }
      }
      waitforperiod(10)
  }
}
```

```
public void run()
{
   while (true)
   {
      ...

      ...

      a.Wait
      ...
      ...

      b.Wait

      synchronized(r1)
      {
       ...
       ...

       }

       ...

       ...

      waitforperiod(20)
   }
}
```

Figure 3.9: Java Code and its Modelisation

11. τ_1 is the only task in the RQ, and it can be executed (since resource r_1 was released by τ_4). At completion it triggers τ_5.

12. τ_5 is executed (over r_1 and r_2) and at completion it releases r_1 and it emits to τ_4 which is lost.

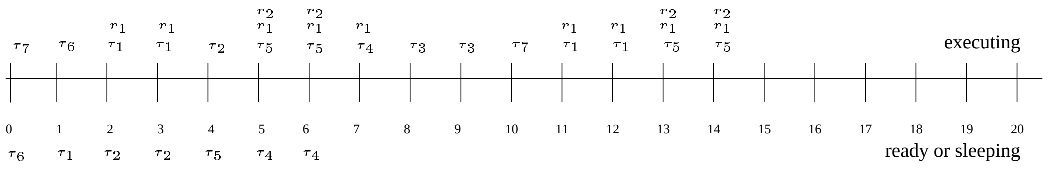


Figure 3.10: Time Line [0,20] for ex.3.5

Example 3.6 *Now suppose the same application as in example 3.4 (where $E_3 = 3$) but both τ_3 and τ_4 use r_1. The system shows the same evolution as before until point 9*

9. At time $t = 8$ τ_4 finishes and triggers τ_3 which begins execution.

10. At $t = 10$ the next period of H_1 arrives and τ_7 preempts τ_3; τ_3 goes to the SQ with r_1 retained and τ_7 executes and the triggers τ_1.

11. τ_1 is the only task in the RQ, and it has higher priority than τ_3 but it cannot be executed since it needs r_1 retained by taui3, waiting at the SQ.

12. τ_3 regains execution finishing at $t = 12$

13. τ_1 executes and finishes at $t = 14$, triggers τ_5 which finishes at $t = 16$.

The time line in figure 3.11 shows the evolution of this example where some kind of priority inversion is due to resource management.

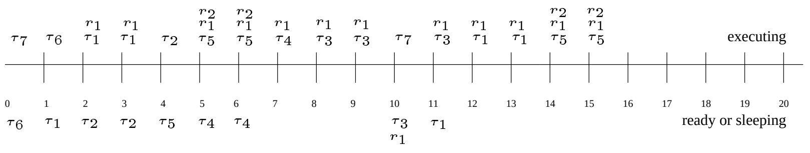


Figure 3.11: Time Line [0,20] for ex.3.6

Example 3.7 *Now suppose an application as shown in figure 3.12, where τ_3 waits for an emit from τ_5.*

1. τ_7 has the highest priority but $\pi_1 < \pi_6$ and $\pi_5 < \pi_4$ due to the await/emit relation.

2. τ_7 begins execution and τ_6 goes to the RQ.

3. τ_7 triggers τ_1 which goes to the RQ and

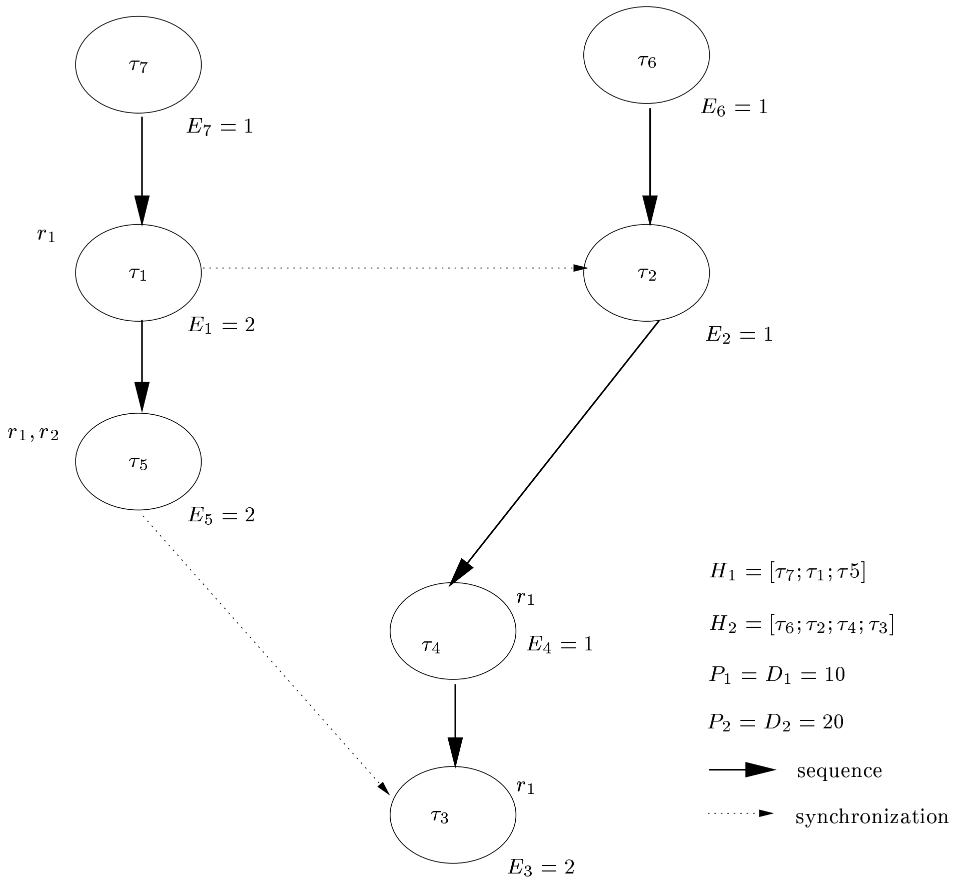


Figure 3.12: Two Threads with shared resources

4. τ_6 is chosen to be executed; at completion it triggers τ_2 which goes to the WQ.
5. τ_1 is executed, setting the lock over r_1 and at its completion it emits to τ_2 and triggers τ_5.
6. τ_2, awaken by τ_1, goes to the RQ and joins τ_5; $\pi_5 > \pi_2$ τ_5 is chosen to be executed, and at completion it emits to τ_3 which is lost; τ_5 releases both r_1 and r_2.
7. τ_2 is executed and triggers τ_4.
8. τ_4 executes over r_1 and when it finishes it triggers τ_3^0 which waits an emit from τ_5 in WQ *retaining* r_1.
9. At $t = 10$ the next period of H_1 arrives and τ_7 is executed, triggering τ_1.
10. τ_1 is the only task in the RQ, but it cannot be executed since it needs r_1 retained by τ_3, waiting in the WQ.
11. τ_3 is also blocked and it will never be awaken. We are in the presence of a *deadlock*.

Figure 3.13 shows the time line.

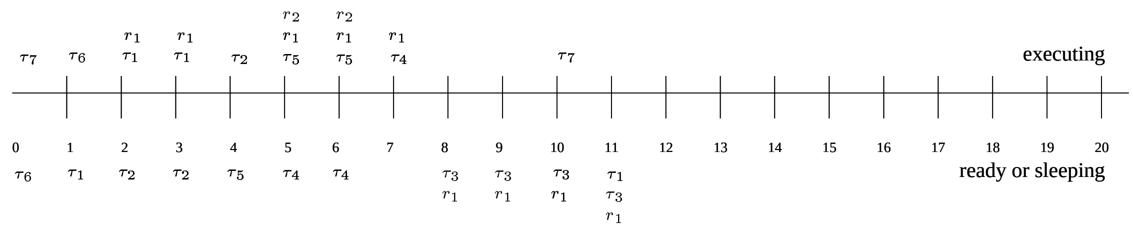


Figure 3.13: Time Line for ex. 3.7

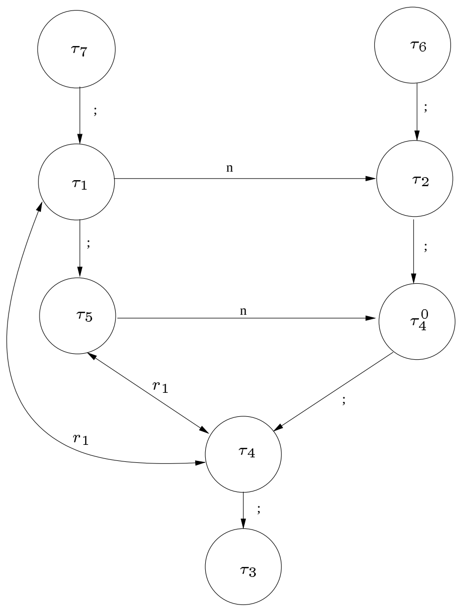


Figure 3.14: Wait for Graph example 3.1

3.4.1 Conflict Graphs

We have shown three examples of scheduling using our policy one of which shows a deadlock, a situation clearly non-schedulable. How can we detect this situation? Are there any structural properties of the system which can lead us to avoid deadlock?

One well stablished algorithm to deal with tasks and shared resources is the PCP or IIP; we could apply these protocols and perform schedulability analysis, using the priorities computed by our 2 rules, 1 and 2. Instead, we propose to analyse the relationships among our tasks and take advantage of their structure.

Example 3.8 *Recall our example 3.1; figure 3.14 illustrates the use of our model as a wait for graph,* WFG*, based on the sequence, (";"), await/emit, ("n") and resource, ("r"), relationships.*

In this graph we can see a cycle among (τ_4, τ_1, τ_2, τ_4^0) and also among (τ_1, τ_5, τ_4) and as usual, cycles in a WFG represent a risk of blocking or deadlock situation. Note that τ_4^0 is an artificial task to mark the difference between τ_4 waiting for an *emit* and τ_4 in the RQ waiting for execution over r_1.

In this graph, we should eliminate those precedence relations which are not harmful: typically the ";″ relation is not harmful because when a task finishes it is "sure" that it triggers its successor task (if any). The problem is in the presence of "n''-arcs or "r''-arcs which risk a task to wait an infinite amount of time.

If we analyse cycle (τ_4, τ_1, τ_2, τ_4^0), we see that τ_4 will wait for the execution of τ_2, but this time is bounded by τ_2's execution; τ_2 waits for an emit from τ_1, which may be lost risking τ_2 from livelock. On the other hand, τ_1 can be blocked by τ_4 if this task is executing (and hence has r_1) but this time is bounded. In a similar manner, τ_4 could be waiting for τ_1 and τ_5 but this time is also bounded. In other words, once τ_4 joins the RQ it has the notify it needs and eventually it can progress as r_1 is unlocked (by τ_5 The other cycle is analysed in a similar manner.

So, this apparent cycles can be pruned if we delete all *safe* wait for relations, we can get a graph without cycles, shown in figure 3.15(a).

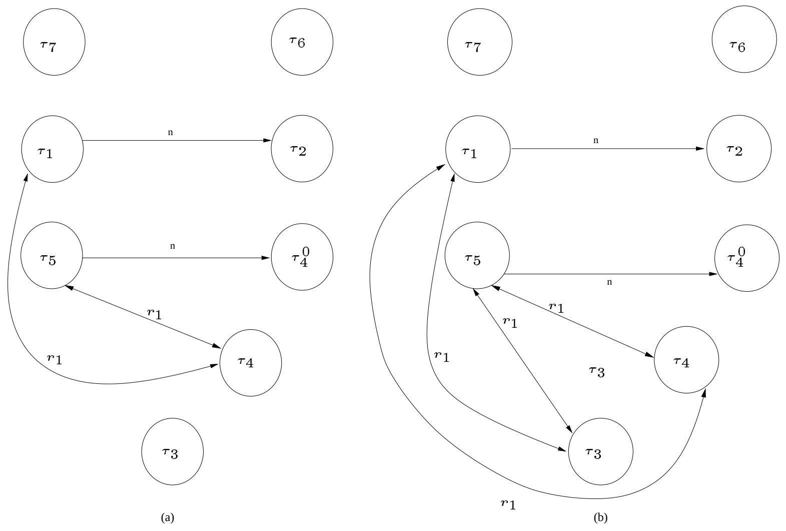


Figure 3.15: Pruned and Cyclic WFG

Example 3.9 *In figure 3.15(b), we show the* WFG *for example 3.6 where we have added two arcs of 'r'-type between τ_1 and τ_3 and τ_5 and τ_3. For simplicity, we have omitted the ';' arcs.*

Even the elimination of the arcs of type ";″ does not provoke the elimination of cycles, but a cycle involving just one resource is not a deadlock. In fact, in our model, if resource r_1 is assigned to τ_1 then τ_5 can also progress and hence release r_1 for τ_4 and τ_3. Analogously if r_1 is assigned to τ_4.

Example 3.10 *In figure 3.16 we show the* WFG *for example 3.7, where there are many cycles but only one involves* two *resources, i.e. r_1 and the* emit *from τ_5 to τ_3 (which can be considered as a resource retained by τ_5.*

In this system the deadlock situation cannot be prevented, since τ_3 waits in the WQ *retaining* r_1 and then preventing τ_1 (and τ_5) to progress; as τ_3 needs an emit from τ_5 the system is in a deadlock situation.

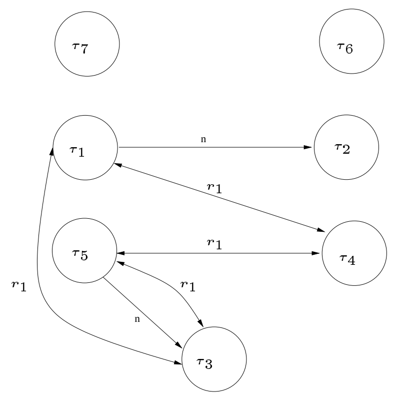


Figure 3.16: Cyclic Wait for Graph

In conclusion, this system is inherently deadlockable under our *fixed priority assignment* and so it is non schedulable, as indicated by the cycle in the corresponding graph involving more than one resource. We could imagine another strategy to handle resources, inserting criteria in the code to create dynamic priorities according to the state of the system.

So, for our priority assignment method, the analysis may be completed by the construction of these conflicts graphs, eliminating those arcs which show a safe wait for relation, that is, arcs showing a sequence of tasks and verifying the existence of cycles which show a deadlock situation. Our method is safe and simple: associating static priorities and verifying cycles assures schedulability but the method is not complete, since we can find other assigments for our non-schedulable systems.

3.4.2 Implementation

Once we have modelled our Java Program and that a possible schedule is found, we must introduce these rules within our code, in order to create a real time Java program.

Our scheduler, based on temporal constraints and await/emit relations can give the following solution to our application example 3.1:

$$\pi_7 = 7, \pi_1 = 3, \pi_5 = 3, \pi_6 = 4, \pi_2 = 4, \pi_4 = 2, \pi_3 = 2$$

Rule 1 partially orders some independent tasks from different threads based on some fixed criteria, such as deadline. We can say $\Pi_1 > \Pi_2$, so task τ_7 has the highest priority; as $\mathcal{N}_1 = \{\tau_1, \tau_5\}$ their priorities are treated by rule 2. Then, $\pi_6 > \pi_1$ but τ_1 must have a priority greater than τ_3 and τ_4 (if we want to keep the priority relation within different periods). Similarly $\pi_2 > \pi_5$, but τ_5 must have priority greater than that for τ_3 and τ_4.

So the scheduler must place these priority relationships in the synchronization points, which consider the whole set of active tasks when a new arrival is produced. We show in figure, 3.17 a possible implementation using the primitive `setpriority` from RT-Java.

```
class PeriodicTh extends Thread
{ long p ;
  ThreadBody b ;
  PeriodicTh(long p, ThreadBody b)
  {   this.p = p ;
      this.b = b ;
  }
  public void run()
  { long t ;
    Clock c = new Clock() ;
    while(true)
    { t = c.getTime() ;
      b.exec() ;
      waitforperiod(p + t - c.getTime());
    }
  }
}

interface ThreadBody
{    public void exec() ;
}

class Thread1_body implements ThreadBody
{ Event a, b ;
  Thread1_body (Event a, b)
  { this.a = a ;
    this.b = b ;
  }
  public void exec()
  { this.setpriority(7);
    t7 ;
    this.setpriority(3) ;
    t1 ;
    a.emit;
    this.setpriority(3) ;
    t5 ;
    b.emit;
  }
}
```

```
class Thread2_body implements ThreadBody
{ Event a, b ;

  public void exec()
  { this.setpriority(4) ;
    t6;
    this.setpriority(4) ;
    a.await;
    t2;
    this.setpriority(2) ;
    t4;
    this.setpriority(2) ;
    b.await;
    t3;
  }
}

class Scheduler
{ public static void main(String argv[])
  { Event a = new Event() ;
    Event b = new Event() ;
    Thread1_body th1_body = new Thread1_body(a,b) ;
    Thread2_body th2_body = new Thread2_body(a,b);
    PeriodicTh thread1 = new PeriodicTh(10, th1_body) ;
    PeriodicTh thread2 = new PeriodicTh(20, th2_body) ;
  }
}

class Event
{ public void emit()
  { synchronized(this) {this.notify}
  }

  public void await()
  { synchronized(this) {this.wait}
  }
}
```

Figure 3.17: Two Scheduled Threads

Chapter 4

Life is Time, Time is a Model

Résumé

Ce chapitre présente les modéles temporels basé sur les automates temporisés et ses extensions. Nous donnons la définition d'un automate temporisé classique et nous continuons avec les automates avec chronomêtres et avec tâches. Dans une deuxième partie nous presentons trois utilisations differentes de ces automates pour attaquer la modélisation.

Layout of the chapter

This chapter deals with models used to abstract RTS and their application to the schedulability problem. The chapter is organized as follows: we introduce timed models, starting by timed automaton and an analysis of a well known problem: reachability; then we continue with some extensions of this machine: timed automata with deadlines, with chronometers and with tasks; finally we show the application of these basic models to the schedulability problem through three approaches: synthesis, task composition and job-shop. No doubt that this chapter only shows a partial state of the art in the theory and evolution of timed automata, guided by our needs and contributions.

4.1 Timed Automata

A *timed automaton*, TA, is a finite state automaton with *clocks*, [10]. A clock is a real time function which records time between events; all clocks advance at the same pace in a monotonously increasing manner and eventually they can be updated to a new value.

Each transition of a TA is a *guarded transition*, that is a predicate, defined over clocks, which if true permits the transition to be taken. A transition may also be decorated by clock update operations.

Formally, a TA **A** is a 5-uple $(\mathcal{S}, \mathcal{C}, \Sigma, \mathcal{E}, I)$, where:

- $\mathcal{S}$ is a set of *states* (s, v), where s is a location and v a valuation of clocks.
- $\mathcal{C}$ is a set of *clocks*.
- Σ is the *alphabet*, a set of labels or *actions*.

- $\mathcal{E}$ is the set of *edges*. Each edge e is a tuple (s, σ, g, μ, s') where
  - $s \in \mathcal{S}$ is the *source* state and $s' \in \mathcal{S}$ is the *target* state.
  - $\sigma \in \Sigma$ is the label.
  - g is the *guard* or *enabling condition* and
  - μ is the *clock assignment*
- I is the *invariant constraint*, defined over clocks; $I(s)$ is the invariant of $s \in \mathcal{S}$.

We need to formalize what we understand as *clock assignment* and an *invariant constraint.*

An *assignment* is a mapping of a clock $c \in \mathcal{C}$ into another clock or 0; the operation of setting a clock to zero is called *reset* operation. The set of assignments over $\mathcal{C}$, denoted $\mathit{\Gamma}_{\mathcal{C}}$, is the set $\{\mathcal{C} \rightarrow \mathcal{C}^*\}$, where $\mathcal{C}^* = \mathcal{C} \cup \{0\}$.

The set of valuations of $\mathcal{C}$, denoted $\mathcal{V}_{\mathcal{C}}$ is the set $[\mathcal{C} \rightarrow \Re^+]$ of total functions from $\mathcal{C}$ to $\Re^+$.

Let $\gamma \in \mathit{\Gamma}_{\mathcal{C}}$, we denote by $v[\gamma]$ the clock valuation such that for all $x \in \mathcal{C}$ we have:

$$v[\gamma](c) = \begin{cases} 0 & \text{if } \gamma(c) \in \mathit{\Gamma}_{\mathcal{C}} \\ v(c) & \text{otherwise} \end{cases}$$

Definition 4.1 (C-Constraint) *A clock constraint or* C-Constraint *is an expression over clocks which follows the grammar:*

$$\Psi ::= x \leq d | x - y \leq d | \psi_1 \wedge \psi_2 | \neg \Psi$$

where $x, y \in \mathcal{C}$ are clocks and $d \in \mathbb{Q}$ is a rational constant.

Invariants and guards are elements of Ψ; invariants are associated to states, that is to each state we associate a formula $I(s) \in \Psi$ and each guard g of an edge $e \in \mathcal{E}$ is also a clock constraint; expressions from Ψ control the transition operations to traverse an edge and the predicate states to remain in a state.

Example 4.1 *Figure 4.1 shows a simple* TA *for a periodic task T_1 with period $P = 10$.*

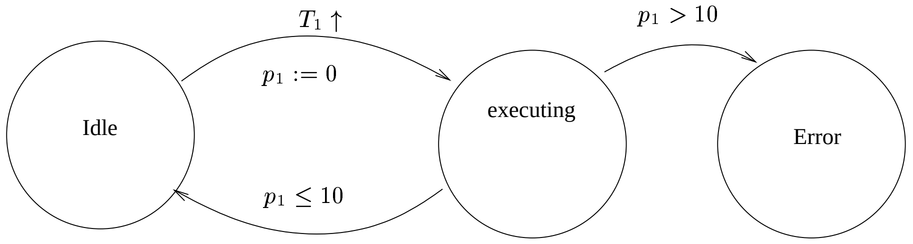


Figure 4.1: Modelling a periodic task

Sometimes it is useful to partition the set Σ into two sets of *controllable* and *uncontrollable* actions, noted Σ^c and Σ^u, respectively. Controllable actions are those actions time independent, which can be known at compile time and often tied to functional aspects of the application, for instance, access to shared resources. Uncontrollable actions are those actions dependent of the environment which may suffer from disturbances, for instance, process arrival, [8].

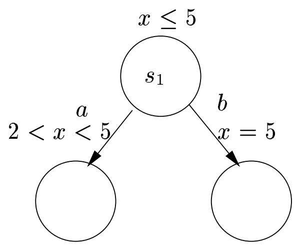


Figure 4.2: Invariants and Actions

The role of invariants. Conditions over states, expressed as a formula in Ψ, allow the specification of hard or soft deadlines: when for some action a deadline is reached, the continuous flow of time is interrupted and the action is forced to occur. We say that and action is then *urgent.* On the contrary, we say that an action is *delayable* if whenever it is enabled, its execution can be postponed by letting time progress; at some time a delayed action may become urgent. In figure, 4.2 we see an example; action a is enabled when clock x attains a value greater than 2; the invariant in s_1 let us remain while $x \leq 5$; at any moment between $(2,5)$ we can execute action a, we say a is delayable. On the contrary, when clock x attains 5 we *must* execute action b, since it is enabled at $x = 5$ but cannot be postponed, we say b is urgent. Sometimes we will mark an edge e with an *urgency type* $\zeta \in \{\delta, \epsilon\}$ for delayable or urgent actions.

Semantics. A TA **A** is then useful to model a transition system $(\mathcal{Q}, \rightarrow)$, where $\mathcal{Q}$ is a set of *states* and $\rightarrow$ is a *transition relation.* A state of **A** is given by a location and a valuation of clocks and a transition is the result of traversing an outgoing edge while respecting the enabling conditions and probably setting clocks according to an assignment.

More precisely, **A** can remain in a location while time passes respecting the corresponding invariant condition; in this case, clocks are updated by the amount of time elapsed; these are called *timed transitions.* When the valuation satisfies the enabling condition of an outgoing edge, **A** can cross the edge, and the valuation is modifed according to the assignment; these are called *discrete* transitions.

Formally, $(\mathcal{Q}, \rightarrow)$ is defined, [60]:

1. $\mathcal{Q} = \{(s, v) \in \mathcal{S} \times \mathcal{V}_{\mathcal{C}} | v \models I(s)\}$, that is, the set of states is composed by pairs of location and clock valuation, implying the invariant condition.
2. The transition $\rightarrow \subseteq \mathcal{Q} \times (\Sigma \cup \Re^+) \times \mathcal{Q}$ is defined by:
   (a) Discrete transitions:
   $$\frac{(s, \sigma, g, \mu, s') \in \mathcal{E} \wedge v \models g \wedge v[\mu] \models I(s')}{(s, v) \xrightarrow{\sigma} (s', v[\mu])}$$
   where $(s', v[\gamma])$ is a *discrete successor* of (s, v); conversely, the latter is the *discrete predecessor* of the former.
   (b) Timed transitions:
   $$\frac{\delta \in \Re^+ \; \forall \delta' \in \Re^+ \; \delta' \leq \delta \Rightarrow (s, v + \delta') \models I(s')}{(s, v) \xrightarrow{\delta} (s, v + \delta)}$$
   where $(s, v + \delta)$ is a *time successor* of (s, v); conversely the latter is said to be a *time predecessor* of the former.

Definition 4.2 (Execution) *An* execution *or* run *r of a timed automaton* **A** *is an infinite sequence of states and transitions:*

$$r = s_0 \overset{l_0}{\to} s_1 \overset{l_1}{\to} \dots$$

where $s_i \in \mathcal{S}$, $l_i \in (\Sigma \cup \Re^+)$ and $i \in \mathbb{N}$.

That is, an execution is the evolution of the automaton according to the events and the time elapsed in the system.

We denote by $\mathcal{R}_{\mathbf{A}}(q)$ the set of runs starting at $q \in \mathcal{Q}$ and by $\mathcal{R}_{\mathbf{A}} = \bigcup_{q \in \mathcal{Q}} \mathcal{R}_{\mathbf{A}}(q)$ the set of runs for **A**.

4.1.1 Parallel Composition

How can we combine two or more timed automata? The *composition* is the combination of timed automata.

Definition 4.3 (Parallel Composition) *Let $\mathbf{A}_i = (\mathcal{S}_i, \mathcal{C}_i, \Sigma_i, \mathcal{E}_i, I_i)$, for $i = 1, 2$ be two* TA *with disjoint sets of locations and clocks. The* **parallel composition** $\mathbf{A}_1 ||_\Sigma \mathbf{A}_2$*, defined over a set of actions Σ is the* TA $(\mathcal{S}, \mathcal{C}, \Sigma, \mathcal{E}, I)$*, where:*

- $\mathcal{S} = \mathcal{S}_1 \times \mathcal{S}_2$,
- $\mathcal{C} = \mathcal{C}_1 \cup \mathcal{C}_2$,
- $I(s) = I_1(s_1) \wedge I_2(s_2)$ *if* $s = (s_1, s_2)$, $s_1 \in \mathcal{S}_1, s_2 \in \mathcal{S}_2$,
- $\mathcal{E}$ *s defined by the following rules:*

$$\frac{e_1 = (s_1, \sigma, g_1, \mu_1, s_1') \in \mathcal{E}_1, e_2 = (s_2, \sigma, g_2, \mu_2, s_2') \in \mathcal{E}_2}{e = ((s_1, s_2), \sigma, g, \mu, (s_1', s_2')) \in \mathcal{E}, g = g_1 \wedge g_2, \mu = \mu_1 \cup \mu_2}$$

$$\frac{e_1 = (s_1, \sigma_1, g_1, \mu_1, s_1') \in \mathcal{E}_1, \sigma_1 \in \Sigma_1 \wedge \sigma_1 \notin \Sigma_1 \cap \Sigma_2}{e = ((s_1, s_2), \sigma_1, g_1, \mu_1, (s_1', s_2)) \in \mathcal{E}}$$

That is for those common actions, we define a common transition as the product of the individual transitions; for each of the non-shared actions, we define a new transition. The second rule is applied symmetrically to the other component.

4.1.2 Reachability

One main problem in Automata Theory is the reachability analysis, that is which are the states reachable from a state q, by executing the automaton, starting at q.

Definition 4.4 (Reachability) *A state q' is* reachable *from state q if it belongs to some run starting at q; we define $\mathsf{Reach}_{\mathbf{A}}(q)$ the set of states reachable from q:*

$$\mathsf{Reach}_{\mathbf{A}}(q) = \{q' \in \mathcal{Q} | \exists r = q_0 \overset{l_0}{\to} q_1 \overset{l_1}{\to} \dots \in \mathcal{R}_{\mathbf{A}}(q), \exists i \in \mathbb{N}, q_i = q'\}$$

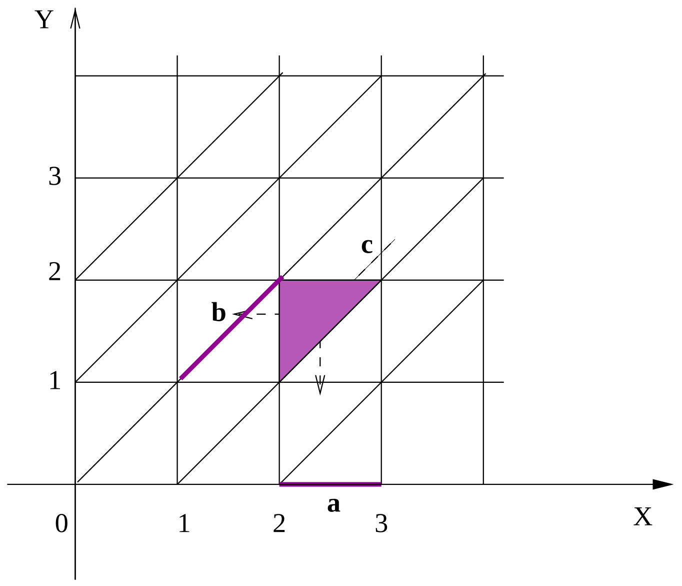


Figure 4.3: Region Equivalence

The problem is how to compute this set; there are many different approaches; we shall use the notion of *region graphs* to develop an algorithm, see [60].

A region is a hypercube characterized by a clock constraint.

Example 4.2 *Figure 4.3 illustrates the concept; a region is defined by the clock constraint* $2 < x < 3 \wedge 1 < y < 2 \wedge x - y < 1$, *marked in grey in the figure.*

Region equivalence

Let $\Psi_{\mathcal{C}}$ be a non-empty set of clock constraints over $\mathcal{C}$. Let $D \in \mathbb{N}$ be the smallest constant which is greater than or equal to the absolute value $|d|$ of every constant $d \in \mathbb{Z}$ appearing in a clock constraint in Ψ. We define $\simeq_{\Psi_{\mathcal{C}}} \subseteq \mathcal{V}_{\mathcal{C}} \times \mathcal{V}_{\mathcal{C}}$ to be the largest reflexive and symmetric relation such that $v \simeq_{\Psi_{\mathcal{C}}} v'$ iff for all $x, y \in \mathcal{C}$, the following three conditions hold:

1. $v(x) > D$ implies $v'(x) > D$
2. if $v(x) \leq D$ then
   (a) $\lfloor v(x) \rfloor = \lfloor v'(x) \rfloor$ and
   (b) $\sqsubseteq(v(x)) = 0$ implies $\sqsubseteq(v'(x)) = 0$, where $\lfloor \cdot \rfloor$ is the integer part function and $\sqsubseteq(\cdot)$ is the fractional part function.
3. for all clock constraints in $\Psi_{\mathcal{C}}$ of the form $x - y \prec d$, $v \models x - y \prec d$ implies $v' \models x - y \prec d$.

$\simeq_{\Psi_{\mathcal{C}}}$ is an equivalence relation and is called the bf region equivalence for the set of clock constraints $\Psi_{\mathcal{C}}$; as usual, we denote $[v]$ the equivalence class of v. Regions can be characterized by a clock constraint and as clocks evolve at the same path, each region is graphically represented as a hypercube with some 45^o diagonals.

Recall Figure 4.3 and let v be any clock valuation in this region.

1. Consider the assignment $y := 0$; the clock valuation under this assignment belongs to the region $2 < x < 3 \wedge y = 0$, marked as **a** in the figure.

2. Consider the assignment $x := y$; this clock valuation $v[x := y]$ belongs to the region $1 < x < 2 \wedge 1 < y < 2 \wedge x = y$, marked as **b**.

3. Finally, if we consider the succesor of v we can see that it belongs to some region crossed by a straight line drawn in the direction of the arrow.

Consider a TA **A** as defined in 4.1 and its transition system $(\mathcal{Q}, \rightarrow)$. We extend the region equivalence $\simeq_{\Psi_C}$ to the states of $\mathcal{Q}$ as follows: two states $q = (s, v)$ and $q' = (s', v')$ are *region equivalent*, denoted $q \simeq_{\Psi_C} q'$ iff $s = s'$ and $v \simeq_{\Psi_C} v'$. We denote by $[q]$ the equivalence class of q.

The region equivalence over states can be stablished as follows:

Definition 4.5 (State Equivalence) *Let $\Psi_{\mathbf{A}}$ be the set of all clock constraints appearing in* **A** *and let $q_1, q_2 \in \mathcal{Q}$ such that $q_1 \simeq_{\Psi_C} q_2$, then:*

1. *For all $\sigma \in \Sigma$, whenever $q_1 \xrightarrow{\sigma} q_1'$ for some q_1' there exists q_2' such that $q_2 \xrightarrow{\sigma} q_2'$ and $q_2 \simeq_{\Psi_C} q_2'$.*

2. *For all $\delta \in \mathbb{R}^+$, whenever $q_1 \xrightarrow{\delta} q_1'$ for some q_1' there exists q_2' and $\delta' \in \mathbb{R}^+$ such that $q_2 \xrightarrow{\delta'} q_2'$ and $q_2 \simeq_{\Psi_C} q_2'$.*

The region equivalence over states is said to be **stable** *with respect to the transition relation $\rightarrow \subseteq \mathcal{Q} \times (\Sigma \cup \mathbb{R}^+) \times \mathcal{Q}$.*

This definition implies that for all region-equivalent states q_1 and q_2, if some state q_1' is reachable from q_1, a region-equivalent state q_2' is reachable from q_2.

Let $\hat{\Psi} \subseteq \Psi_C$ be a set of clock constraints, $\hat{\Psi}_{\mathbf{A}}$ be the set of clock constraints of **A**, and $\simeq$ be the region equivalence defined over $\hat{\Psi} \cup \hat{\Psi}_{\mathbf{A}}$. Let $\tau \notin \Sigma$ and let $\Sigma_\tau = \Sigma \cup \{\tau\}$.

Definition 4.6 (Region-Graph) *The region graph* $\mathtt{R}(\mathbf{A}, \hat{\Psi})$ *is the transition system $(\mathcal{Q}_\simeq, \rightarrow)$ where:*

1. $\mathcal{Q}_\simeq = \{[q] \mid q \in \mathcal{Q}\}$

2. $\rightarrow \subseteq \mathcal{Q}_\simeq \times \Sigma_\tau \times \mathcal{Q}_\simeq$ is such that:

   (a) for all $\sigma \in \Sigma$ and for all ρ, $\rho' \in \mathcal{Q}_\simeq, \rho \xrightarrow{\sigma} \rho'$ iff there exists $q, q' \in Q$ such that $\rho = [q], \rho' = [q']$, and $q \xrightarrow{\sigma} q'$.

   (b) for all ρ, $\rho' \in \mathcal{Q}_\simeq, \rho \xrightarrow{\tau} \rho'$ iff

      i. $\rho = \rho'$ is an unbounded region or,

      ii. $\rho \neq \rho'$ and there exists $q \in Q$ and a real positive number δ such that $q \xrightarrow{\delta} q'$ and $\rho = [q], \rho' = [q + \delta]$, and for all $\delta' \in \mathbb{R}^+$, if $\delta' \leq \delta$ then $[q + \delta']$ is either ρ or ρ'.

We define $\texttt{Reach}(\rho)$ to be the set of regions reachable from the region ρ as

$$\texttt{Reach}(\rho) = \{\rho' | \rho \to^* \rho'\}$$

where $\to^*$ is the reflexive and transitive closure of $\to$.

We denote by $\langle q \rangle$ any clock constraint $\psi \in \Psi$ such that $q \models \psi$ and for all $\psi' \in \Psi$, if $q \models \psi'$ then ψ implies ψ'. That is, $\langle q \rangle$ is the tightest clock constraint that characterizes the values of the clocks in q. The question whether the state q' is reachable from the state q can be answered using the following property:

Property 4.1 (Reachability) *let* $\mathbf{A}$ *be a* TA, $q, q' \in \mathcal{Q}$ *and let* $\mathtt{R}(A, \{\langle q \rangle, \langle q' \rangle\})$ *be the corresponding region graph, then:*

$$q' \in \texttt{Reach}(q) \text{ iff } [q'] \in \texttt{Reach}([q])$$

The constraints $\langle q \rangle$ and $\langle q' \rangle$ characterize exactly the equivalence classes $[q]$ and $[q']$ respectively.

4.1.3 Region graph algorithms

The basic idea of the algorithm using the region graph concept is the use of property `Reachability` as shown in the previous section. Two ways of answering whether q' is reachable from q are *forward traversal* and *backward traversal*

The first starts from a state q and by visiting its succesors, and the successors of those and so on, until we find q' in some region or all regions have been visited; in summary, we need a sequence of regions $F_0 \subseteq F_1 \subseteq \ldots$, such that:

$$\begin{aligned} F_0 &= [q] && (4.1) \\ F_{i+1} &= F_i \cup \mathrm{Suc}(F_i) && (4.2) \end{aligned}$$

where $\mathrm{Suc}(F_i) = \{\rho \mid \exists \rho_i \in F_i . \, \rho_i \to \rho\}$

Property 4.2 (Forward Reachability) *For all* $q, q' \in \mathcal{Q}, [q'] \in \texttt{Reach}([q]) \text{iff} [q'] \in \bigcup_{i \geq 0} F_i$

The second approach starts from a state q', visits its predecessors, and the predecessors of those and so on, until the state q is found or all regions have been visited. Similarly, we construct a sequence of regions $B_0 \subseteq B_1 \ldots$ such that:

$$\begin{aligned} B_0 &= [q'] && (4.3) \\ B_{i+1} &= B_i \cup \mathrm{Pre}(B_i) && (4.4) \end{aligned}$$

where $\mathrm{Pre}(B_i) = \{\rho \mid \exists \rho_i \in B_i . \, \rho \to \rho_i\}$

Property 4.3 (Backward Reachability) *For all* $q, q' \in \mathcal{Q}, [q'] \in \texttt{Reach}([q]) \text{iff} [q] \in \bigcup_{i \geq 0} B_i$

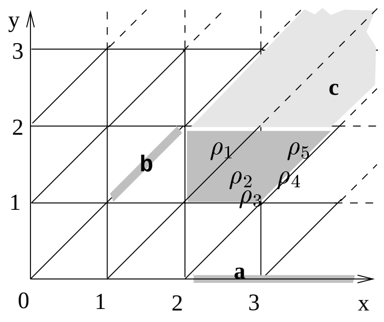


Figure 4.4: Representation of sets of regions as clock constraints

4.1.4 Analysis using clock constraints

Let F be the set of regions $\bigcup_{i\geq 0} F_i$ computed by the forward traversal algorithm explained in Section 4.1.3. Then F can be symbolically represented as a disjoint union of the form $\biguplus_{s\in\mathcal{S}} F_s$, where F_s is the clock constraint that characterizes the set of regions that belong to F whose location is equal to s. The same observation holds for B. Indeed, such characterization can be computed without a-priori constructing the region graph.

4.1.5 Forward computation of clock constraints

Let $s \in \mathcal{S}$, $\psi_s \in \Psi_{\mathcal{C}}$ and $e = (s, \sigma, g, \mu, s') \in \mathcal{E}$. We denote by $\mathsf{Suc}_e(\psi_s)$ the predicate over $\mathcal{C}$ that characterizes the set of clock valuations that are reachable from the clock valuations in ψ_s when the timed automaton executes the discrete transition corresponding to the edge e. That is,

$$v \models \mathsf{Suc}_e(\psi_s) \quad \text{iff} \quad \exists v' \in \mathcal{Q}.\ v = v'[\gamma] \wedge v' \models (\psi_s \wedge \psi).$$

Property 4.4 $\mathsf{Suc}_e(\psi_s) \in \Psi_{\mathcal{C}}$.

Example 4.3 *Consider again the example illustrated in Figure 4.4. Recall that ψ is the clock constraint $1 < y < 2 \wedge 2 < x \wedge x - y < 2$.*

a. *The result of executing the transition resetting x to 0 is computed as follows.*

$$\begin{array}{ll} \mathsf{Suc}_a(\psi_s) = & \\ = & \exists x', y'.\ \psi_s[x/x', y/y'] \wedge y = 0 \wedge x = x' \\ = & \exists x', y'.\ 1 < y' < 2 \wedge 2 < x' \wedge x' - y' < 2 \wedge y = 0 \wedge x = x' \\ = & \exists y'.\ 1 < y' < 2 \wedge 2 < x \wedge x - y' < 2 \wedge y = 0 \\ = & 2 < x \wedge x < 4 \wedge y = 0 \end{array}$$

Since the upper bound of 4 is greater than the constant $C = 3$, we can eliminate the clock constraint $x < 4$ and obtain: $\mathsf{Suc}_a(\psi_s) = 2 < x \wedge y = 0$.

b. *Now, consider the assignment $x := y$.*

$$\begin{aligned}
&\mathit{Suc}_{\mathsf{b}}(\psi_s) = \\
&= \exists x', y'.\ \psi_s[x/x', y/y'] \wedge y = y' \wedge x = y' \\
&= \exists x', y'.\ 1 < y' < 2 \wedge 2 < x' \wedge x' - y' < 2 \wedge y = y' \wedge x = y' \\
&= \exists x'.\ 1 < y < 2 \wedge 2 < x' \wedge x' - y < 2 \wedge x = y \\
&= 1 < y < 2 \wedge 0 < y \wedge x = y \\
&= 1 < y < 2 \wedge x = y
\end{aligned}$$

In other words, to compute $\mathsf{Suc}_e(\psi_s)$ is equivalent to visit all the regions that are e-successors of the regions in ψ_s, but without having to explicitly represent each one of them.

Let $s \in \mathcal{S}$ and $\psi_s \in \Psi_{\mathcal{C}}$. We denote by $\mathsf{Suc}_\tau(\psi_s)$ the predicate over $\mathcal{C}$ that characterizes the set of clock valuations that are reachable from the clock valuations in ψ_s when the timed automaton lets time pass at s. That is,

$$v \models \mathsf{Suc}_\tau(\psi_s) \quad \text{iff} \quad \exists \delta \in \mathbb{R}^+.\ v - \delta \models \psi_s \wedge \forall \delta' \in \mathbb{R}^+.\ \delta' \leq \delta \Rightarrow v - \delta' \models I(s).$$

Property 4.5 *$\mathit{Suc}_\tau(\psi_s) \in \Psi_{\mathcal{C}}$.*

Example 4.4 *Consider again the example illustrated in Figure 4.4. Case* c *corresponds to letting time pass at the location. For simplicity, we assume here that the invariant condition is true.*

$$\begin{aligned}
&\mathit{Suc}_\tau(\psi_s) = \\
&= \exists \delta \in \mathbb{R}^+.\ \psi_s[x/x - \delta, y/y - \delta] \\
&= \exists \delta \in \mathbb{R}^+.\ 1 < y - \delta < 2 \wedge 2 < x - \delta \wedge (x - \delta) - (y - \delta) < 2 \\
&= \exists \delta \in \mathbb{R}^+.\ 1 < y - \delta < 2 \wedge 2 < x - \delta \wedge x - y < 2 \wedge \\
&= 1 < y \wedge 2 < x \wedge y - x < 0 \wedge x - y < 2
\end{aligned}$$

Notice that $\mathsf{Suc}_\tau(\psi_s)$ characterizes the set of the regions that contains the regions characterized by ψ_s and the regions reachable from them by taking only τ-transitions.

Now, we can solve the reachability problem by computing the sequence of sets of clock constraints $F_0, F_1, \cdots$ as follows:

$$\begin{aligned}
F_0 &= \langle q \rangle \\
F_{i+1} &= \biguplus_{s \in \mathcal{S}} \left(\mathsf{Suc}_\tau(F_{i,s}) \ \uplus \ \biguplus_{e \in \mathcal{E}} \mathsf{Suc}_e(F_{i,s}) \right)
\end{aligned}$$

Notice that $F_{i,s}$ implies $F_{i+1,s}$ for all $i \geq 0$ and $s \in \mathcal{S}$.

Property 4.6 *Let $F = \bigcup_{i \geq 0} F_i$, $q = (s, v)$, and $q' = (s', v')$. $[q'] \in \mathit{Reach}([q])$ iff $\langle q' \rangle$ implies $F_{s'}$.*

4.2 Extensions of TA

From the classic definition of TA, there have been developed many variations, principally regarding the nature of clock operations; we will present some extensions of TA used to model RTS.

4.2.1 Timed Automata with Deadlines

A TAD is a tuple $(\mathcal{S}, \mathcal{C}, \Sigma, \mathcal{E}, D)$ where $\mathcal{S}, \mathcal{C}, \Sigma$ and $\mathcal{E}$ are defined as for TA and $D : \mathcal{E} \to \Psi$, associates with each edge $e \in \mathcal{E}$ a *deadline* condition specifying when the edge e is urgent. For $s \in \mathcal{S}$ we define

$$D(s) = \bigvee_{e=(s,\sigma,g,\mu,s')\in\mathcal{E}} D(e)$$

and we define

$$I(s) = \neg D(s)$$

which shows that TAD behaves like a TA where time can progress at a location as long as all the deadline conditions associated with the outgoing edges are not satisfied.

The difference between TA and TAD is the addition of a deadline condition for edges; for a given edge e, its guard g_e determines when e *may* be executed, while $D(e)$ determines when it *must* be executed; that is the guard is a kind of enabling condition while a deadline is an urgency conditions. Clearly, for all the states satisfying $\neg g_e \wedge D(e)$, time can be blocked and it is reasonable to require $D(e) \models g_e$ to avoid time deadlocks. When $D(e) = g_e$ e is immediate and must be executed as soon as it becomes enabled. If $D(e)$ is false, e is delayable at any state.

We can now afford the operation of composition for TAD, in which the resulting TAD has the same structures for $\mathcal{S}, \mathcal{C}, \Sigma, \mathcal{E}$ as mentioned for composition for TA except for deadlines which follow the rules:

$$\frac{e_1 = (s_1, \sigma, g_1, \mu_1, s_1') \in \mathcal{E}_1, e_2 = (s_2, \sigma, g_2, \mu_2, s_2') \in \mathcal{E}_2, \sigma \in \Sigma}{e = ((s_1, s_2), \sigma, g, \mu, (s_1, s_2')) \in \mathcal{E}, g = g_1 \wedge g_2, \mu = \mu_1 \cup \mu_2, D = D_1 \wedge D_2}$$

$$\frac{e_1 = (s_1, \sigma_1, g_1, \mu_1, s_1') \in \mathcal{E}_1, \sigma_1 \in \Sigma_1 \wedge \sigma_1 \notin \Sigma_1 \cap \Sigma_2}{e = ((s_1, s_2), \sigma_1, g_1, \mu_1, (s_1', s_2)) \in \mathcal{E}, D = D_1}$$

4.2.2 Timed Automata with Chronometers

As seen in the definition of TA, clocks may be assigned a value from $\Re$; sometimes it is useful to offer a richer set of operations over clocks. We present in this section, two variants of TA: *stopwatch automaton* and *updatable timed automaton*.

Stopwatch Automaton

Classic TA operates over clocks through the operation of reset or more generally the operation of *set*: $x := d$ where x is a clock and d a constant from $\mathbb{Q}$. Clocks evolve at the same constant pace, that is, for all clocks its derivative is 1.

A variant of TA is a stopwatch automaton, SWA, where clocks can be suspended; McMannis et al, [40] propose a SWA where the rate of increase or derivative of a clock can be set to 0. Later, a clock can be unsuspended to resume increasing at rate 1. Kesten et al, [31] propose a hybrid automaton where the derivative of a clock can be set to any constant from the set of integers.

The basic definition of a SWA is the same as that for TA except that we add a relation *rate* to each location associated to the clocks in that location and their behaviour, stopped or running.

A SWA is a tuple $\mathbf{A}_{\text{SWA}} = (\mathcal{S}, \mathcal{C}, \Sigma, \mathcal{R}, \mathcal{E}, I)$ where $\mathcal{S}, \mathcal{C}, \Sigma$ and I are defined as for TA and

- $\mathcal{R} : \mathcal{C}^{\mathcal{S}} \to \{0, 1\}^N$, associates to each clock $c_i \in \mathcal{C}$ in state $s_j \in \mathcal{S}$ a rate value of 0 or 1. If c_i is a clock running in state s_j then $r_i^j = 1$, otherwise it is 0.

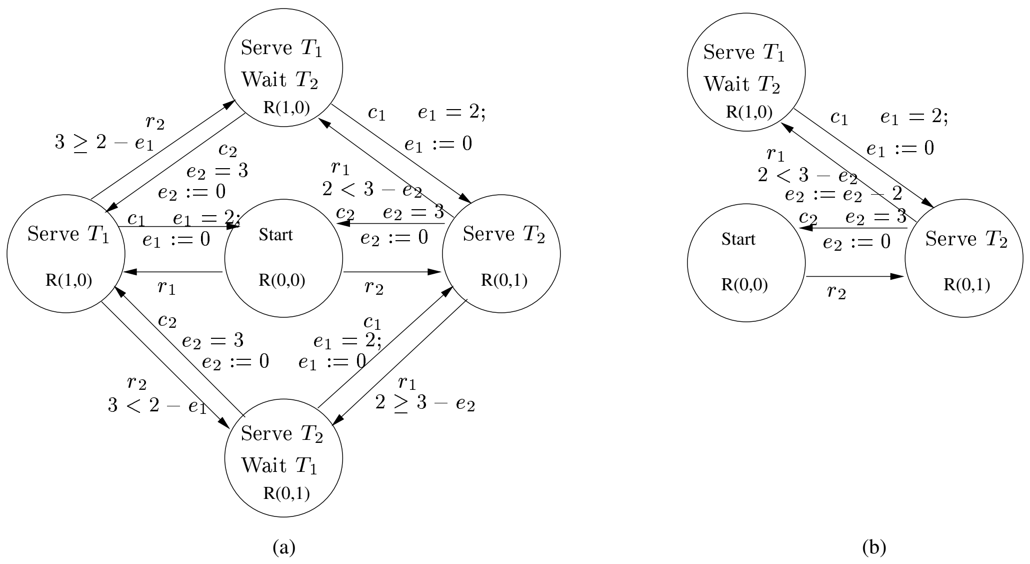


Figure 4.5: Using SWA and UTA to model an application

- $\mathcal{E}$ is also modified by an update operation, i.e, clocks may be reset, and also be decremented by some fixed rational constant; if $e \in \mathcal{E}$ is the tuple (s, σ, g, μ, s'), then s, σ, g and s' are as defined for TA and μ is the clock *update*, including the reset operation (denoted $c_i := 0$) and the *decrement operation* of the form $c_i := c_i - d$, where $c_i \in \mathcal{C}$ and $d \in \mathbb{Q}$..

SWA are very useful for modelling and analysing RTS:

Example 4.5 *Consider two tasks $T_1(2,8), T_2(3,4)$ where numbers in parentheses represent the execution time, E_i and the minimal interarrival time, P_i respectively, for each task T_i, $i \in \{1,2\}$. The application runs under a* least time remaining *policy, that is, the processor performs the task requiring the least amount of time to complete.*

Figure 4.5(a) shows the stopwatch automaton modelling this application where each location represents the status of the task in the system: waiting for service, executing or not requested. For each task T_i, we have a timer e_i accumulating the computed time and the expression $E_i - e_i$ represents the remaining computing time which serves as a priority decision criteria. Clocks are stopped when the corresponding task is not executing. Events $r_i, i \in \{1,2\}$ represent the arrival ot task T_i, and c_i their completion.

Unfortunately, the untimed language of a suspension automata is not guaranteed to be w-regular and some tricks may be introduced to replace the suspension by a decrementation, as we see in the section 4.2.3.

Timed Automata with tasks

A TA with tasks, TAT, is a TA where each location represents a task. The model was originally developped by Fersman et al., [26]; in that paper they call it *extended timed automata.*

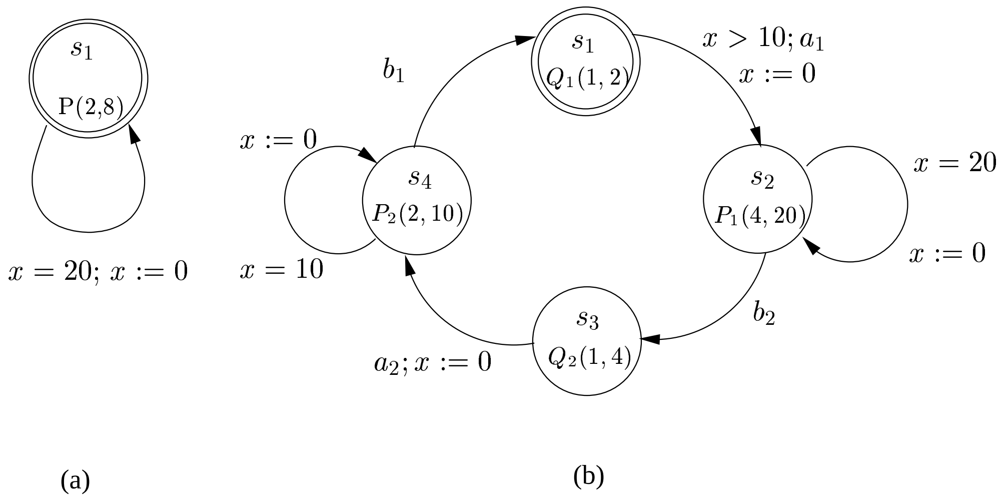


Figure 4.6: Timed Automata Extended with tasks

Definition 4.7 *A* timed automata with tasks $\mathbf{A}_T$ *is a tuple*

$$(\mathcal{S}, \mathcal{C}, \Sigma, \mathcal{E}, s_0, I, T, M)$$

where $\mathcal{S}$, $\mathcal{C}$, Σ, $\mathcal{E}$, I represent the set of states, the clocks, the alphabet over actions, the edges and the invariants as already defined for TA*; we distinguish $s_0 \in \mathcal{S}$ the* initial state, *T the* set of tasks of the application *and $M : \mathcal{S} \hookrightarrow T$, a partial function associating to each location a task.*

M is a partial function, since at some locations, there may be no task associated, since the system is idle.

Example 4.6 *Figures 4.6 shows an example of an* TAT*; in (a) we see a single periodic task $P(2,8)$ with computing time 2 and period 8; in (b) we see four tasks: $P_1(4,20)$ and $P_2(2,10)$ two periodic tasks and $Q_1(1,2)$ and $Q_2(1,4)$ two sporadic tasks triggered by events b_1 and b_2 respectively, both with computing time 1 and minimal interarrival times 2 and 4, respectively.*

Let $\mathbb{P} = \{P_1, P_2, \ldots, P_m\}$ denote the universal set of tasks, periodic or sporadic; each $P_j, 1 \leq j \leq m$ characterized by its pair (E_j, D_j) execution time and deadline, respectively.

From an operational point of view, a TAT represents the currently active tasks in the system; a semantic state $(s, v[c], q)$ gives for a state s the current values of clocks and a queue q, where q has the form $[T_1(e_1, d_1), T_2(e_2, d_2), \ldots, T_n(e_n, d_n)]$ where $T_i(e_i, d_i), 1 \leq i \leq n$ denotes an active instance of task P_j with remaining computing time e_i and remaining time to deadline d_i. T_1 is the current executing task.

A *discrete transition* will result in a new queue sorted according to a scheduling policy, including the recently arrived task. A *timed transition* of δ units implies that the remaining computation time of T_1 is decreased by δ; if this value becomes 0, then T_1 is removed from the queue; all deadlines are decreased by δ. Formally:

Definition 4.8 *Given a scheduling strategy* `Sch` *the semantics of a* TAT $\mathbf{A}_T$ *as given in definition 4.7 with initial state $(s_0, v[c_0], q_0)$ is a transition system defined by the following rules:*

- *Discrete transition over an action σ:*

$$(s, v[c], q) \xrightarrow{\sigma}_{\texttt{Sch}} (s', \mu[c \mapsto 0], \texttt{Sch}(q \oplus M(s'))) \text{ if } s \overset{g,\sigma,c\mapsto 0}{\longrightarrow} s' \wedge \mu \models g$$

where $\mu[c \mapsto 0]$ indicates those clocks, within μ-assignment, to be reset (the others keep their values as time does not diverge), $\oplus$ is the insertion of $M(s')$ in q and Sch *is the sorting of a queue according to a scheduling policy.*

- *Timed transition over δ units of time:*

$$(s, v[c], q) \xrightarrow{\delta}_{\texttt{Sch}} (s, v[c] + t, \texttt{run}((q, \delta))) \text{ if } (v[c] + t) \models I(s)$$

where $\texttt{run}(q, \delta)$ *is a function which returns the transformed queue after δ units of time of execution.*

Remark Observe that q contains two *variables* (*not clocks*), for each active task: the pair (e_i, d_i); as time diverges, these values are updated conveniently to show this evolution; for example if $q = [(5,9),(3,10)]$ and time diverges for 3 units, then we have $q' = [(2,6),(3,7)]$, that is all deadlines are *also* reduced by δ, *but* the value of $e_i, i > 1$ remains unchanged. The next example shows what happens if $\delta \geq e_1$.

Example 4.7 *Consider once again, the example in figure 4.6(b); consider a scheduling policy* EDF*, the following is a sequence of typical transitions*

$$\begin{array}{rcl}
(s_0, [x=0], [Q_1(1,2)]) & \xrightarrow{1} & (s_0, [x=1], [Q_1(0,1)]) \equiv (s_0, [x=1], []) \\
& \xrightarrow{10} & (s_0, [x=11], []) \\
& \xrightarrow{a_1} & (s_2, [x=0], [P_1(4,20)]) \\
& \xrightarrow{2} & (s_2, [x=2], [P_1(2,18)]) \\
& \xrightarrow{b_2} & (s_3, [x=2], [Q_1(1,4), P_1(2,18)]) \\
& \xrightarrow{0.5} & (s_3, [x=2.5], [Q_1(0.5,3.5), P_1(2,17.5)]) \\
& \xrightarrow{a_2} & (s_4, [x=0], [Q_1(0.5,3.5), P_2(2,10), P_1(2,17.5)]) \\
& \xrightarrow{1.5} & (s_4, [x=0.5], [P_2(1,8.5), P_1(2,16)]) \\
& \ldots &
\end{array}$$

We should note two important points shown in this example:

- The first concerns the fact that while in state s_2 or s_4, an infinite number of instances of P_1 or P_2 may arrive, with 20 or 10 units of delay. No deadline is missing, since at the arrival of a new instance, the old one had already finished.
- The queue may potentially grow but it is considerably emptied in state s_1 where we have to wait for more than 10 units before considering event a_1. In fact discrete transitions make the queue grow while timed transitions shrink it.

4.2.3 Timed Automaton with Updates

The model presented for SWA was slightly modified to avoid the operation of stopping a clock, retaining the *update* operation to decrement a clock by a constant from $\mathbb{N}$; this model is known as *updatable timed* *automaton*, UTA in the literature, though the original paper called it *automaton with decrementation*. Nicollin et al, [45] and Bouyer et al, [18], analysed some interesting properties of this class of TA.

An UTA is a tuple $(\mathcal{S}, \mathcal{C}, \Sigma, \mathcal{E})$ where $\mathcal{S}, \mathcal{C}, \Sigma$ are defined as for TA and $\mathcal{E}$ changes in its μ to include the general *set* operation.

Example 4.8 *Regaining our exemple 4.5, we could modify the model, using a* UTA, *where instead of stopping e clocks when the corresponding tasks are preempted, we let them continue running and their values are decremented by the execution time of the terminating task each time a task completes. For instance, (see figure 4.5(a)), while serving T_2, T_1 arrives and if its remaining time is smaller than T_2's, then $T_1 \dagger T_2$ and e_2 is stopped; instead, we could decrement e_2 by the preemption time, E_1, (see figure 4.5(b)) and when resuming T_2 it will have the "true" value. Another solution is to let e_2 diverge and when T_2 resumes, we set $e_2 := e_2 - E_1$. In both cases, the effect is the same; some care should be taken in the first solution if we do not want clocks to be negative.*

4.3 Difference Bound Matrices

We present in this section a data structure which is commonly used to implement some of the algorithms of reachability analysis: *difference bound matrices*, DBM [22]

Let $\mathcal{C} = \{c_1, \cdots, c_n\}$, and let $\Lambda \subseteq \Psi_{\mathcal{C}}$ be the set of clock constraints over $\mathcal{C}$ defined by conjunctions of constraints of the form $c_i \prec c$, $c \prec c_i$ and $c_i - c_j \prec c$ with $c \in \mathbb{Z}$. Let u be a clock whose value is always 0, that is, its value does not increase with time as the values of the other clocks. Then, the constraints in Λ can be uniformly represented as bounds on the difference between two clock values, where for $c_i \in \mathcal{C}$, $c_i \prec c$ is expressed as $c_i - u \prec c$, and $c \prec c_i$ as $u - c_i \prec -c$.

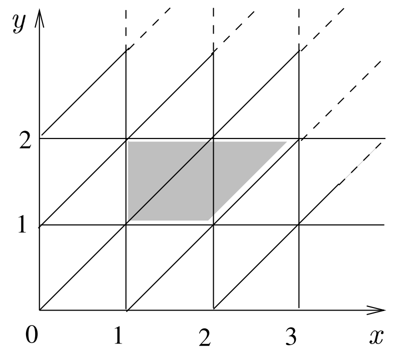
Such constraints can be then encoded as a $(n+1) \times (n+1)$ square matrix D whose indices range over the interval $[0, \cdots, n]$ and whose elements belong to $\mathbb{Z}_\infty \times \{<, \leq\}$, where $\mathbb{Z}_\infty = \mathbb{Z} \cup \{\infty\}$. The first column of D encodes the upper bounds of the clocks. That is, if $c_i - u \prec c$ appears in the constraint, then D_{i0} is the pair $(c, \prec)$, otherwise it is $(\infty, <)$ which says that the value of clock c_i is unbounded. The first row of D encodes the lower bounds of the clocks. If $u - c_i \prec -c$ appears in the constraint, D_{0i} is $(-c, \prec)$, otherwise it is $(0, \leq)$ because clocks can only take positive values. The element D_{ij} for $i, j > 0$, is the pair $(c, \prec)$ which encodes the constraint $c_i - c_j \prec c$. If a constraint on the difference between c_i and c_j does not appear in the conjunction, the element D_{ij} is set to $(\infty, <)$.

Note that for all elements (i, j) an upper bound $M_{i,j}$ is given for the difference $c_i - c_j$ between clocks c_i and c_j. During symbolic state space exploration we are interested in computing the *future* of M, and we need to take into account which clocks are stopped and which are running. Clearly if c_i and c_j are both stopped, both running or only c_i is stopped, then the bound $M_{i,j}$ remains valid; if only c_j is stopped, the difference may grow to ∞; values in M need to be in a canonical form, where all bounds $M_{i,j}$ are as tight as possible

Example 4.9 *Let Λ be the clock constraint $1 < y < 2 \wedge 1 < x \wedge x - y < 2$. Figure 4.7a shows its matrix representation.*

Remark Every region can be characterized by a clock constraint, and therefore be represented by a DBM.

As a matter of fact, many different DBM's represent the same clock constraint. This is because some of the bounds may not be *tight* enough. As already mentioned, values in M need to be as tight as possible, [20, 40, 60]



a

$$\begin{array}{c|ccc} D & c_0 & x & y \\ \hline c_0 & (0,\leq) & (-1,<) & (-1,<) \\ x & (\infty,<) & (0,\leq) & (1,<) \\ y & (2,<) & (\infty,<) & (0,\leq) \end{array}$$

b

$$\begin{array}{c|ccc} D' & c_0 & x & y \\ \hline c_0 & (0,\leq) & (-1,<) & (-1,<) \\ x & (3,<) & (0,\leq) & (1,<) \\ y & (2,<) & (1,<) & (0,\leq) \end{array}$$

Figure 4.7: Representation of convex sets of regions by DBM's.

Example 4.10 *Consider again the clock constraint depicted in Figure 4.7. The matrix* b *is an equivalent encoding of the clock constraint obtained by setting the upper bound of x_1 to be $(3,<)$ and the difference $x_2 - x_1$ to be $(1,<)$. Notice that this two constraints are implied by the others.*

However, given a clock constraint in Λ, there exists a *canonical* representative. Such a representative exists because pairs $(c, \prec) \in \mathbb{Z}_\infty \times \{<, \leq\}$, called *bounds*, can be ordered. This induces a natural ordering of the matrices. Bounds are ordered as follows. We take $<$ to be strictly less than $\leq$, and then for all $(c, \prec), (c', \prec') \in \mathbb{Z}_\infty \times \{<, \leq\}$, $(c, \prec) \leq (c', \prec')$ iff $c < c'$ or $c = c'$ and $\prec \leq \prec'$. Now, $D \leq D'$ iff for all $0 \leq i, j \leq n$, $D_{ij} \leq D'_{ij}$.

Example 4.11 *Consider the two matrices in Figure 4.7. Notice that $D' \leq D$.*

For every clock constraint $\psi \in Cnd$, there exists a unique matrix C_ψ that encodes ψ and such that, for every other matrix D that also encodes ψ, $C_\psi \leq D$. The matrix C_ψ is called the canonical representative of ψ and can be obtained from any matrix D that encodes ψ, by applying to D the Floyd-Warshall [6] algorithm [22, 59, 46, 60] for details. We will always refer to a DBM to mean the canonical representative where bounds are tight enough.

Encoding convex timing constraints by DBM's requires then $\mathcal{O}(n^2)$ memory space, where n is the number of clocks. Several algorithms have been proposed to reduce the memory space needed [17, 33].

The verification algorithms require basically six operations to be implemented over matrices: conjunction, time successors, reset successors, time predecessors, reset predecessors and disjunction. These operations are implemented as follows.

Conjunction. Given D and D', $D \wedge D'$ is such that for all $0 \leq i, j \leq n$, $(D \wedge D')_{i,j} = \min(D_{ij}, D'_{ij})$.

Time successors. As time elapses, clock differences remain the same, since all clocks increase at the same rate. Lower bounds do not change either since there are no decreasing clocks. Upper bounds have to be pushed to infinity, since an arbitrary period of time may pass. Thus, for a canonical representative D, $\mathsf{Suc}_\tau(D)$ is such that:

$$\mathsf{Suc}_\tau(D)_{ij} = \begin{cases} (\infty,<) & \text{if } j = 0, \\ D_{ij} & \text{otherwise.} \end{cases}$$

Reset successors. First notice that resetting a clock to 0 is the same as setting its value to the value of u, that is, $\gamma(c_i) = 0$ is the same as $\gamma(c_i) = u$. Now, when we set the value of c_i to the value of c_j, c_i and c_j become equal and all the constraints on c_j become also constraints on c_i. Having this in mind, the matrix characterizing the set of reset-predecessors of D by reset γ consists in just copying some rows and columns. That is, the matrix $D' = \mathsf{Suc}_\gamma(D)$ is such that for all $0 \leq i, j \leq n$, if $\gamma(c_i) = c_j$ then $\mathsf{row}_i(D') = \mathsf{row}_j(D)$ and $\mathsf{col}_i(D') = \mathsf{col}_j(D)$. [1]

Time predecessors. To compute the time predecessors we just need to push the lower bounds to 0, provided that the matrix is in canonical form. Thus, for a canonical representative D, $\mathsf{Pre}_\tau(D)$ is such that:

$$\mathsf{Pre}_\tau(D)_{ij} = \begin{cases} (0, \leq) & \text{if } i = 0, \\ D_{ij} & \text{otherwise.} \end{cases}$$

Reset predecessors. Recall that the constraint characterizing the set of predecessors is obtained by substituting each clock c_i by $\gamma(c_i)$. Now suppose that we have two constraints $x_k - x_l < c_{kl}$ and $x_r - x_s < c_{rs}$ and we substitute x_k and x_r by c_i, and x_l and x_s by c_j. Then, we obtain the constraints $c_i - c_j < c_{kl}$ and $c_i - c_j < c_{rs}$ which are in conjunction, and so $c_i - c_j < \min(c_{kl}, c_{rs})$. Thus, the matrix $D' = \mathsf{Pre}_\gamma(D)$ is such that for all $0 \leq i \leq n$, $D'_{ij} = \min\{D_{kl} \mid \gamma(x_k) = c_i \wedge \gamma(x_l) = c_j\}$.

Disjunction. Clearly, the disjunction of two DBM's is not necessarily a DBM. That is, Λ is not closed under disjunction, or in other words, the disjunction of two constraints in Ψ is not convex. Usually, the disjunction of D and D' is represented as the set $\{D, D'\}$. Thus, a lot of computational work is needed in order to determine whether two sets of DBM's represent the same constraint.

4.4 Modelling Framework

The process of modelling requires specification of each of the components (tasks) drawn from building blocks fully characterized by their constraints. The operation of *composition* is a key of modelling, since each component is plugged to the system, interacts with other components, represents some code and must respect its (timing) constraints.

To compose a system we can start from a single component, adding other interacting components, so that the obtained system satisfies a given property. This *integration* approach establishes a basic rule for composition which says that if a property P holds for a component C, then this property must be preserved in the composed system. Formally, if $||$ notes composition, if $C \vDash P$ then $C||C' \vDash P$. This assures correctness from construction, unfortunately in general, time dependent properties are non composable, [8, 9].

Another approach to composability, which does not oppose to integration is *refinement*, that is once we have an abstract description of a component T we get a more restricted one T' which verifies if $T \vDash P$ then $T' \vDash P$; normally T' is obtained from T by restricting some observability criteria and a basic rule for composition says that if we replace a component T_i in a composition $T_1|| \ldots T_i \ldots$ by its refinement T'_i, then the new system $T_1|| \ldots T'_i \ldots$ should be a refinement of the initial system.

A *timed model* is essential to the process of synthesis; these models are obtained by adding *time variables*, used to measure the time elapsed, to an untimed model. The natural extension of finite state machines to timed machines is TA and they are a general basic model adopted to face this problem,

[1]Recall that $\gamma(\cdot)$ is a total function.

[51]. A TA is a transition system which evolves through actions (events) or through time steps which represent time progress and uniformly increase time variables.

Composition of timed models is a natural extension of composition of untimed ones, but some care must be taken into account since *clocks evolve at the same rate*, that is, time diverges at the same derivative for all clocks. Furthermore, for timed steps, a synchronous composition rule is applied as a direct consequence of the assumption about a global notion of time.

In general, RTS are modelled through, [51]:

- A timed model for each task
- A Synchronization layer
- A Scheduler

Timed model for tasks To create a timed model for an application, we need to create timed models for its building blocks, generally termed as tasks. For each task, we need to know its resources, a sequence of atomic actions with their execution time (a worst case analysis, in general, or an interval timing constraint with lower and upper bounds) and their timing constraints. We have shown an approach in chapter 3.

Synchronization layer The correctness of the whole application depends on the correctness of each of its components (tasks) but also on the interaction among them. Some kind of synchronization is needed, through the use of primitives to resolve task cooperation and resource management.

We need to differentiate two types of synchronization: timed and untimed. Untimed synchronization is based on the idea that tasks cooperate among them in some kind of producer/consumer model: the output of a task (or of an atomic action within a task) is needed as input for another task. In general, if C_1 and C_2 are two components, then $C_1||C_2$ is the untimed synchronized composition of both tasks. But this composition is not enough, we need some timed extension of this composition to consider timing constraints and hence build the timed synchronized model. Once again, if we have the timed model of two tasks C_1^T and C_2^T, then $C_1^T||^T C_2^T$ represents its timed composition, which includes, of course, the untimed synchronization.

Many problems have been encountered for this approach:

1. Does the timed composed system preserve the main functional properties of the corresponding untimed one?
2. Does the composed system respect some essential properties such as deadlock freedom, liveliness and well timedness?
3. How does the implementation react in front of the timed model? It is worth to note that in a model the reaction to some external stimuli does not take time while the implementation does.
4. Which are the effects of interleaving? It has been shown, [16], that independent actions of untimed components may interleave and cause a (potential) indefinite waiting of a component before it achieves synchronization; the correponding timed system may suffer deadlock, even if the untimed one is deadlock-free.

Scheduler A scheduler has a challenging mission: assure coordination of execution of all system activities to meet timing and QoS requirements. A scheduler interacts with the environment and with the internal execution. Altisen et al, [8] consider a scheduler as a controller of the system model composed of timed tasks with their synchronization and of a timed model of the external environment.

As systems evolve, the role of the scheduler becomes more and more complicated. For independent tasks, the scheduler is simply an arbiter which dispatches tasks in some previously fixed order. As tasks are dependent the scheduler must know the internal state of active tasks in order to take a decision. Finally, for timed tasks the scheduler must know the (timed) internal state of active tasks, and also the behaviour of the environment to decide which task to select.

4.5 A framework for Synthesis

Recall that a timed automaton $\mathbf{A}$ is a 5-uple $(\mathcal{S}, \mathcal{C}, \Sigma, \mathcal{E}, I)$, where $\mathcal{S}$ is a collection of states, $\mathcal{C}$ is a collection of clocks, Σ is an alphabet or set of actions, $\mathcal{E}$ is a set of edges and I is a collection of invariants associated to states.

Once we have the timed model for a task, how do we create a timed model for an application? The basic idea is to use the parallel composition, explained in part 4.1.1.

Sifakis et al, [53] propose a general framework for compositional description using a variant of TA, called *timed automata with deadlines*, TAD, where invariants are replaced by deadline conditions, expressing that if some timing constraints are enabled, then the transition **must** be executed, (see section 4.2.1). TAD are not more (or less) powerful than TA, but the operation of composition over TAD is simpler than in TA.

4.5.1 Algorithmic Approach to Synthesis

One approach to construct a scheduler SCH consists in defining a scheduling policy, as we have seen in chapter 2, that is, we define a systematic way of ordering the execution of a set of tasks, based on some timing constraints, but independently of the application.

The traditional approach to scheduling is suitable for RTS where the behaviour of the environment is not predictable and reactions to external stimuli must be immediate. Altisen et al, [7] proposed a model useful in RTS where the application strongly interacts with the environment, such as multimedia or telecommunications systems. For such systems it is desirable to generate an ad-hoc scheduler at compile time that makes optimal use of the underlying execution hardware and shared resources, guided by knowledge of all possible behaviours of the environment.

A key concept to this approach is the distinction between *controllable* and *uncontrollable* actions. A controllable action corresponds to a transtion that can be triggered by the scheduler and hence known in advance at design time. An uncontrollable action is subject to timing constraints imposed by the environment, which is constantly evolving.

The semantics of the application is given by a timed model $\mathcal{A}$ and a property Q to be satisfied; the method constructs a new timed model $\mathcal{A}_Q$ which models all the behaviours of $\mathcal{A}$ that satisfy Q for any possible sequence of uncontrollable transitions. In summary, quoting Altisen et al, [7]

> ... $\mathcal{A}_Q$ describes all the schedules that satisfy the property, a schedule being a sequence of controllable transitions for a given pattern of uncontrollable behaviours

Typically, the synthesis algorithm is applied to properties of the form $\Box P$, read as "always P". Initially we start with states that satisfy P and keep on iterating over a single step controllable predecessor operator **pre** until a fixed point is reached:

```
$Q^0 = P$
repeat
   $Q^{i+1} = Q^i \cap \mathbf{pre}(Q^i)$
until $Q^i = Q^{i+1}$
```

thus obtaining Q^*. Given a predicate P of a state, the operator **pre** represents all the states of the timed model from which it is possible to reach a state of P by taking some controllable transition, possible after letting time pass, while ensuring that there is no uncontrollable transition that leads into $\neg P$.

Let Q be a property and $Q^* = \bigcup_{s \in S} Q^*_s$ be the set of states computed by the algorithm above. The timed model $\mathcal{A}_Q$ has the same structure as $\mathcal{A}$ and the same timing information, except for its controllable guards, since each guard g_e has been replaced by $g'_e = g_e \wedge Q^*_s \wedge \mathbf{pre}_a(Q^*_{s'})$ where $\mathbf{pre}_a(P)(s,c) = P(s', v(c)) \wedge g(v)$, while uncontrollable transitions remain unchanged.

Example 4.12 *We present an example from [7] to illustrate the application of the synthesis algorithm for reachability properties, see figure 4.8. A multimedia document is composed of six tasks: music[30,40], video[15,20], audio[20,30], text[5,10], applet[20,30] and picture [20,∞] where each task is characterized by its execution interval.*

In the begining, music, video, audio and applet are launched in parallel and we have the following synchronization constraints:

1. *video and audio terminate as soon as any one of them ends; their termination is immediately followed by the text to be displayed;*
2. *music and text must terminate at the same time;*
3. *the applet is followed by a picture;*
4. *the document terminates as soon as both the picture and the music (and text) have terminated;*
5. *the execution times of both the audio and the applet depend on the machine load and are therefore uncontrollable.*

Development Clock x controls music, y video, audio and text, and z applet and picture. For each application, the corresponding guard is $E^m \leq c \leq E^M$, where c is its associated clock and E^m, E^M the minimal and maximal duration time. The finishing condition is $g^c = (30 \leq x \wedge 5 \leq y \wedge 20 \leq z \wedge 20 \leq x - y \leq 35 \wedge x - z \leq 40 \wedge y - z \leq 10)^c$ obtained by a process described in [7].

The idea is to seek for the existence of a scheduler that moves the system from the *initial* state to the state *done*. The property is $\Diamond$*done* and the result obtain is that the document is indeed schedulable. The execution time of *text* can be dynamically adapted to the duration of *video* and *audio* so as to make *music* and *text* terminate synchronously. The corresponding scheduler is shown in figure 4.8, where the restricted guards of controllable transitions, computed by the syntehsis algorithm, are printed in bold. Notice that if *video* terminates at time $y < 20$, the marking {music, text, applet} will be reached with a valuation satisfying $x - y < 20$ wihich falsifies the synchronization guard g^c and therefore the only possible schedule guaranteeing the reachability of *done* must terminate *video* at $y = 20$.

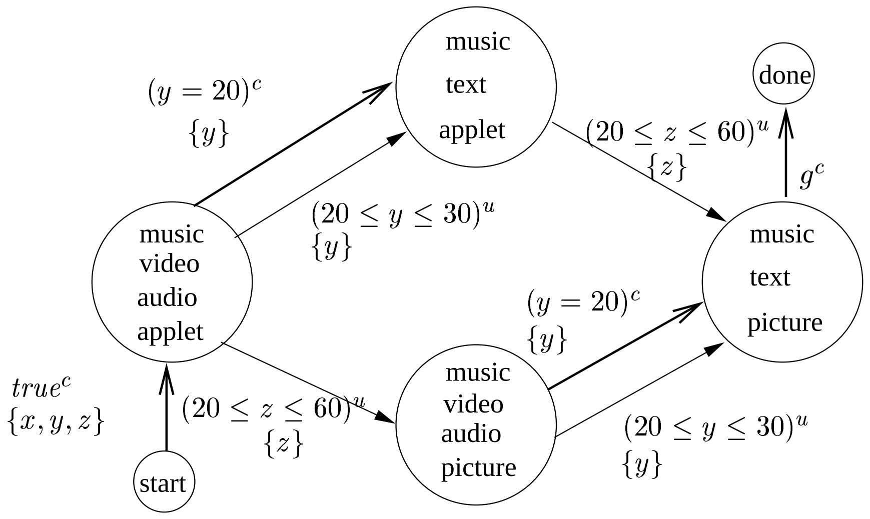


Figure 4.8: Synthesis using TAD

4.5.2 Structural Approach to Synthesis

Altisen et al, [9, 8] have proposed a different methodology based on the construction of a scheduler tailored to the particular application and regardless of any a priori fixed scheduling policy, since we are considering not only the set of tasks but also the behaviour of the environment and some non functional properties such as QoS.

There exists some theoretical methodology for the construction of scheduled systems, [8] based on:

1. A functional description of the processes to be scheduled, as well as their resources and the associated synchronization;
2. Timing requirements added to the functional description which relate execution speed with the external environment;
3. Requirements for the scheduling algorithm:
   (a) Priorities: fixed or dynamic (for pending requests of the processes),
   (b) Idling: a scheduler may not satisfy a pending request due to higher priority requests and
   (c) Preemption: a process of lower priority is preempted when a process of higher priority raises a request.

Taking into account these model constraints, we can follow a methodology for constructing a scheduled system and a timed specification of the process to be scheduled, [8], based on control invariants and their composability and the scheduling requirements expressed as constraints, (some of which are indeed invariants).

The idea is to decompose the global controller synthesis procedure into the application of simpler steps. At each step *a control invariant* corresponding to a particular class of constraints is applied to further restrict the behaviour of the system to be scheduled. The scheduler is decomposed into:

1. Global Scheduling: characterized by a constraint K of the form $K = K_{algo} \wedge K_{sched}$ where K_{algo} specifies a particular scheduling algorithm and K_{sched} expresses schedulability requirements of the processes
2. Computation of control invariants: at each step the corresponding control invariant is computed in a straightforward manner.
3. Iteration: the scheduled system can be obtained by successive applications of steps restricting the process behaviour by control invariants implying all the scheduling constraints, but some composability conditions must be satisfied.

Each constraint is a state predicate represented as an expression of the form $\bigvee_{i=1}^{n} s_i \wedge \psi_i$ where $\psi_i \in \Psi$ is a C-constraint, (an expression over clocks), and s_i is the boolean denoting presence at state s_i.

Given a timed system TS and a constraint K, the *restriction* of TS by K denoted as TS/K is the timed system T where each guard g_e of a controllable transition is replaced by

$$g'_e = g_e \wedge K(s', \mu_0)$$

where μ_0 is the set of clocks reset in e.

In a restricted system TS/K, the C-constraint K is a *control invariant* of TS if $TS/K \models \mathrm{inv}(K)$, that is K is preserved by edges all long the execution of the transition system.

The problem of synthesis was defined by Altisen et al, [8] as:

Definition 4.9 (Synthesis Problem) *Solving the* **synthesis problem** *for a timed system TS and a constraint K amounts to giving a non-empty control invariant K' of TS which implies K, $K' \Rightarrow K, TS/K' \models inv(K')$.*

We need a scheduling requirement expressed as a C-constraint, K. If K' is acontrol invariant implying K then TS/K' describes as scheduled system.

To clarify these concepts we present an example:

Example 4.13 *Let us model a periodic non-preemptable process PP of period $P > 0$, execution time E and relative deadline $D(0 < E \leq D \leq P)$.*

In figure, 4.9 we illustrate our example and we can distinguish three states, **s**leeping, **w**aiting and **e**xecuting; the actions a, b and f stand for arrive, begin and finish; timer x is used to measure the execution time, while timer t measures the time elapsed since process arrival; both timers progress uniformely; b is the only controllable action and guards g are decorated with an *urgency type.* Notice that since the transition b is delayable, the processor may wait for a non-zero time even if processor is free.

Consider a timed system TS = TS$_1$||TS$_2$ where TS$_1$ and TS$_2$ are instances of the periodic process shown in figure 4.9, with parameters (E, P, D) equal to (5,15,15) and (2,5,5) for process 1 and 2 respectively. We can create the constraint:

$$\begin{aligned} K_{\text{dlf}} = \quad & [(s_1 \wedge t_1 \leq 15) \vee (u_1 \wedge x_1 \leq 5 \wedge t_1 \leq 15) \vee (w_1 \wedge t_1 \leq 10)] \wedge \\ & [(s_2 \wedge t_2 \leq 5) \vee (u_2 \wedge x_2 \leq 2 \wedge t_2 \leq 5) \vee (w_2 \wedge t_2 \leq 3)] \end{aligned}$$

which expresses the fact that each one of the processes is deadlock-free: from a control state, time can progress to enable the guard of some exiting transition. This constraint is a proper invariant for TS.

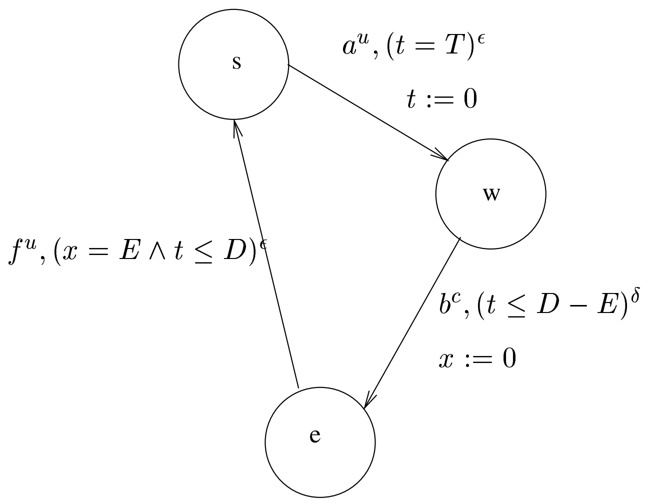

s
w
e
$a^u, (t = T)^\epsilon$
$t := 0$
$b^c, (t \leq D - E)^\delta$
$x := 0$
$f^u, (x = E \wedge t \leq D)^\epsilon$

Figure 4.9: A periodic process

Priorities Priorities are necessary in modelling formalisms for RTS since there may be urgent processes or they may be useful as a conflict resolution mechanism by associating priorities to states or more generally, specifying a state constraint and an associated priority order. Priorities are intrinsically related to preemption policies.

In [9] priorities are defined as a strict partial order over the actions. Formally, a priority order is a strict partial order $\prec \subseteq A^c \times A$ and we say that if $a_1 \prec a_2$, then a_1 must be done before a_2.

Altisen et al, have proved that the application of a priority rule to a timed system respecting conflicting actions through a partial priority order defines a new timed system.

Example 4.14 *In figure 4.10 we see a part of the composed automaton for TS, where some conflict exists between b_1 and b_2 as both access a common resource. Then from the control state (w_1, w_2) of the composed system, the priority rule:*

$$\pi = (D_i - (t_i + E_i) < D_j - (t_j + E_j)), b_j \prec b_i$$

where $(i, j) \in \{(1, 2), (2, 1)\}$ expresses the rule for conflict resolution; the guards of b_1 and b_2 can be conveniently modified as shown in figure 4.10, (note action b is still controllable but the transition is immediate).

Conflict resolution and hence priorities are defined according to a scheduling policy SCH; in our example, we have chosen the least laxity first, [44] which is a mixture of EDF and remaining executing times.

4.6 Schedulability through TAT

In this section, we discuss another approach to modelling, introduced by Fersman et al, [26, 25] and first discussed in [24]. The main idea of the model is to offer a schedulability frame, for a set of non-periodic tasks, triggered by external stimuli, relaxing the general assumption of considering their

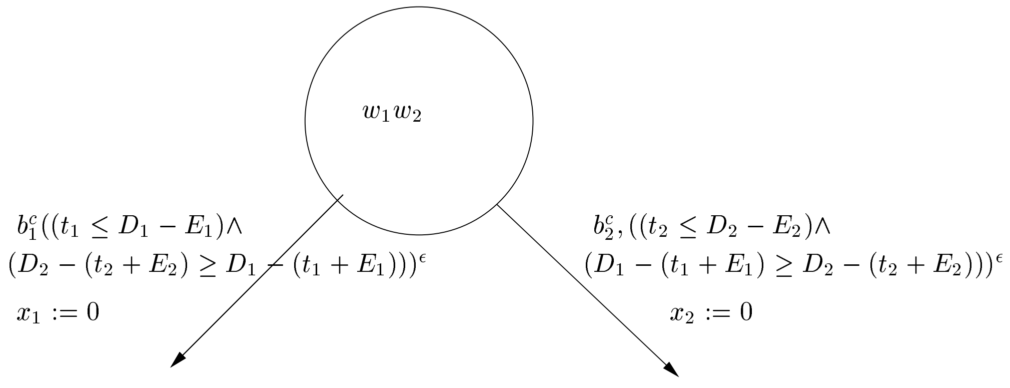

$w_1 w_2$
$b_1^c((t_1 \leq D_1 - E_1)\wedge$
$(D_2 - (t_2 + E_2) \geq D_1 - (t_1 + E_1)))^\epsilon$
$x_1 := 0$
$b_2^c, ((t_2 \leq D_2 - E_2)\wedge$
$(D_1 - (t_1 + E_1) \geq D_2 - (t_2 + E_2)))^\epsilon$
$x_2 := 0$

Figure 4.10: Priorities

minimal interarrival times as task periods, as this analysis is pessimistic in many cases and indeed it does not take into account the evolution of the environment.

To model the application, TAT are used, (see section 4.2.2), where each state of the automaton corresponds to a task; a transition leading to a location in the automaton denotes an event triggering a new task and the guard on the transition specifies the possible arrival times of the event; clocks may be updated by the decrementation operations shown in 4.2.3. A state of such an automaton includes not only the location and the clock assignment but also a queue q which contains pairs of remaining computing times and relative deadlines for all active tasks.

Task set is denoted $\mathbb{P} = \{P_1, P_2, \ldots, P_m\}$, where each task $P_j, 1 \leq j \leq m$ is characterized by a pair (E_j, D_j), as usual. The active set of tasks is $T = \{T_1, T_2, \ldots, T_n\}$ where each $T_i \in \mathbb{P}, 1 \leq i \leq n$; the system may accept many instances of the same task P_j, in which case they are copies of the same program with different inputs[2].

4.6.1 Schedulability Analysis

Remember that a TAT is a transtion system characterized by triples of the form $(s, v[c], q)$ where s is a state, $v[c]$ values of clocks in s and q a queue of tasks sorted by some scheduling policy. The notion of schedulability is then transposed to q: if all tasks in q can be computed within their deadlines, the system is schedulable and hence an *automaton is schedulable* if all reachable states of the automaton are schedulable. Two important results are drawn out from this model:

1. Under the assumption of **non-preemptive scheduling policies**, the schedulability checking problem can be transformed to a reachability problem for TAT and thus it is **decidable**.

2. Under the assumption of **preemptive scheduling policies**, a conjecture was made over the *undecidability* of the schedulability checking problem, since preemptive scheduling is associated with stop-watch automata for which the reachability problem is undecidable. This conjecture was proved as wrong if UTA are used, (recall that in UTA clocks may be updated by substraction), and if clocks are upper bounded and substraction leaves clocks in the bounded zones; the reachability problem is then **decidable**.

[2]sometimes P_i is called a *task type* and we distinguish instances as $T_i^1, T_i^2, \ldots$

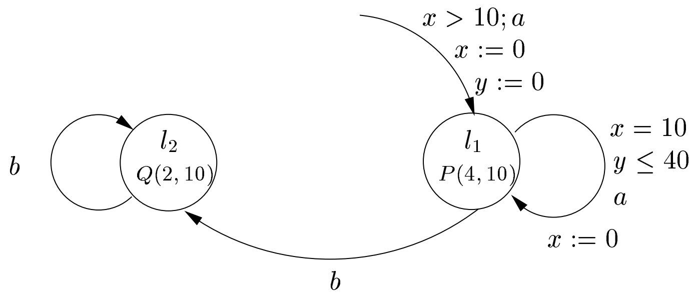


Figure 4.11: Zeno-behaviour

The schedulability problem may be reduced to the problem of location reachability as for normal TA not considering task assignment, abstracting from the extended model; with this analysis we can check properties such as safety, liveliness or many others not related to the task queue.

However as properties to the task queue are of interest, Fersman et al, [26] have developped a new verification technique. One of the most intersting properties of TAT related to the task queue, is schedulability. In fact, invariants in location and guards on edges rule the problem of schedulability.

Consider, for example, a part of an TAT shown in figure 4.11; while in location l_1 the system could accept a new event a each 10 units ($x \geq 10$) but no more than 4, due to the constraint $y \leq 40$; in fact, each time a new instance of P arrives, the previous one had already been executed, so the task queue is bounded (by 1 in this case).

On the contrary if we observe state l_2 we see than an infinite number of Q instances could be accepted since the discrete transition b is not guarded, i.e. not constraint by some clocks. This behaviour is not desirable and is called the *zeno behaviour*. Fersman et al have proved that this behaviour corresponds, of course, to non-schedulability as the scheduler cannot manage to finish an infinite number of tasks within a finite time (deadline). We also note that zenoness is a necessary condition for schedulability but not a sufficient condition, since we can easily find a system non-zeno which is not schedulable.

The following definition relies schedulability and reachability.

Definition 4.10 (Schedulability) *A state $(s, v[c], q)$ of an* TAT *is a failure denoted $(s, v[c], \texttt{Error})$ if there exists a task within q which fails to meet its deadline, i.e, if $q = [T_1(e_1, d_1), \ldots, T_n(e_n, d_n)]$, then from an initial state $(s_0, v[c_0], q_0) \xrightarrow{*}_{\texttt{Sch}} (s, v[c], \texttt{Error}) \implies \exists i, 1 \leq i \leq n, s.t.\ e_i > 0 \wedge d_i < 0$ for a given scheduling policy* Sch*.*

In Fersman's methodology, value e_i computing the remaining executing time decreases as task τ_i is executed while values d's computing the remaining time to reach deadlines decrease; under this context, schedulability can be checked by verifying that at any instant t :

$$\sum_{i \leq k} e_i \leq d_k \ \forall\ 1 \leq k \leq n \tag{4.5}$$

which assures that the waiting time for task τ_k, given by the sum of the execution times of tasks with higher priority (according to Sch) is "small enough" to let τ_k finish its execution[3]. Sometimes we can decompose expression 4.5 as

[3]Remember that tasks in q are ordered, being T_1 under execution

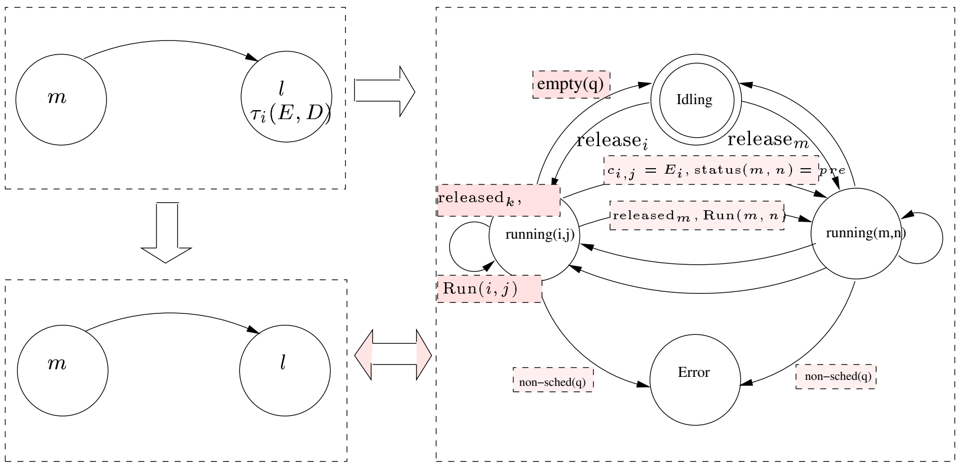


Figure 4.12: Encoding Schedulability Problem

$$\sum_{i\leq k} e_i \leq d_k = \underbrace{\sum_{i<k} e_i}_{B_k} + e_k \leq d_k \ \forall\ 1 \leq k \leq n \tag{4.6}$$

and B_k is called the *blocking time* for task τ_k.

A very important result in Fersman's model is that **the problem of checking schedulability relative to a preemptive fixed priority scheduling strategy for TAT is decidable**.

This result is based on the following ideas, (see figure 4.12):

- For UTA the reachability problem is **undecidable** and hence the reduction of schedulability to reachability is also undecidable.
- A *bounded updatable automata* is a UTA in which each clock c is not negative and bounded by a maximal constant $\mathbb{C}_c$, that is all operations leave clocks non-negative and clocks do not grow beyond a known constant.
- The reachability problem for bounded updatable automata is decidable and hence the schedulability problem, [18, 40].
- They prove that TAT are in the class of bounded UTA.

To encode the problem of schedulability as reachability, Fersman et al, develop a methodology based on three transformation steps:

1. The application is first encoded as a TAT $\mathbf{A}_T$ as we have seen in the precedent paragraphs, where states represent a task, (possibly in execution).
2. $\mathbf{A}_T$ is transformed to a TA $\mathbf{A}$ reduced to actions triggering tasks.
3. Given a scheduling fixed priority strategy Sch, a TAT $\mathbf{A}_{\mathtt{Sch}}$ is developed which includes all tasks and all possible transitions according to priorities. $\mathbf{A}_{\mathtt{Sch}}$ is a UTA, with the following characteristics:

- There are three types of locations: `Idling`, `Running`(i, j) and `Error`, with `Running` being parametrized by a task τ_i and its instance j.
- For each task instance, we have two *clocks*: c_i to denote accumulated computing time since T_i^j was started and $r_{i,j}$ denoting the time since T_i^j was released; c_i is a substracted clock, substraction is applied to note the evolution of time, while T_i is temporarily suspended. For instance, c_i (initially reset to 0) is reduced by δ if task T_k is executed δ units of time and T_k preempts T_i. It is this transformation which moves to the "risking zone of undecidability".

4. The third step of the encoding is to construct the product automaton $\mathbf{A}_{\mathtt{Sch}}||\mathbf{A}$ where both automata synchronize over identical action symbols.

Fersman et al. prove that clocks of $\mathbf{A}_{\mathtt{Sch}}$ are bounded and non negative in the product automaton; for this automaton the reachability analysis of the error state is decidable and equivalent to declare the system as non-schedulable.

For this approach, the number of clocks needed in the analysis is proportional to the maximal number of schedulable task instances associated with a model, which in many cases is huge. In a later paper, [25], Fersman et al prove that for a *fixed priority scheduling strategy*, the schedulability checking problem can be solved by reachability analysis on standard TA using only two extra clocks in addition to the clocks used in the original model to decribe task arrival times.

4.7 Job-Shop Scheduling

To conclude this chapter, we introduce another model for tasks, the *job-shop scheduling problem*, JSS, suitable for distributed systems, under certain conditions, [1].

The JSS problem is a generic resource allocation problem in which machines are required at various time points for given durations by different tasks. Each job J is characterized by a sequence of steps $(m_1, d_1), (m_2, d_2), \dots, (m_k, d_k)$ where $m_i \in M$ and $d_i \in \mathbb{N}, 1 \leq i \leq k$, M being the universal set of machines indicating the required utilization of machine m_i for time duration d_i. The sequence states a logical order to accomplish job J, first machine m_1 for d_1 units of time, then machine m_2 for d_2 time, and so on.

Formally:

Definition 4.11 (Job-Shop Specification) *Let M be a finite set of machines. A job specification over M is a triple $J = (k, \mu, d)$ where $k \in \mathbb{N}$ is the number of steps in J, $\mu : \{1 \dots k\} \rightarrow M$ indicates which resource is used at each step, and $d : \{1 \dots k\} \rightarrow \mathbb{N}$ specifies the length of each step. A job-shop specification is a set $\mathcal{J} = \{J^1, \dots, J^n\}$ of jobs with $J^i = (k^i, \mu^i, d^i)$.*

The model assumes that:

- A job can wait an arbitrary amount of time between two steps, (there is no notion of deadline).
- Once a job starts to use a machine, it cannot be preempted until this step terminates, (that is, there is no preemption).
- Machines are used in a mutual exclusion manner (while job J is using a machine, no other can have access simultaneously) and steps of different jobs using different machines can execute in parallel.

Definition 4.12 (Feasible Schedule) *A feasible schedule for a job-shop specification $\mathcal{J} = \{J^1, \ldots, J^n\}$ is a relation $S \subseteq \mathcal{J} \times K \times \mathbb{R}^+$, so that a triple (i, j, t) from S indicates that job J^i is busy doing its j^{th}-step at time t and hence occupies machine μ^i in its j step. A* feasible schedule *should satisfy the following conditions:*

1. *Ordering: if (i, j, t) and $(i, j', t') \in S$ then $j < j' \longrightarrow t < t'$*
2. *Every step is executed continously until completion.*
3. *Mutual Exclusion: for every $i, i' \in \mathcal{J}, j, j' \in K$ and $t \in \mathbb{R}^+$ if (i, j, t) and $(i', j', t) \in S$, then $\mu^i(j) \neq \mu^{i'}(j')$, two steps of different jobs which execute at the same time do not use the same machine.*

The optimal JSS problem is to find a schedule with the shortest length over t over all $(i, j, t) \in S$.

4.7.1 Job-shop and TA

Naturally, each job $J = (k, \mu, d)$ can be modeled as a TA such that for each step j where $\mu(j) = m$ we create a state indicating the use of m for a duration of d, but we have to mark also the waiting time *before* using m; for this reason, [1] proposes to create states $\bar{m}$ for each machine used by J.

We will not give the formal definition of this transformation, but illustrate it through an example.

Example 4.15 *Consider two jobs $J^1 = \{(m_1, 4), (m_2, 5)\}$ and $J^2 = \{(m_1, 3)\}$, over $M = \{m_1, m_2\}$. The automata corresponding to these jobs is shown in figure 4.13(a), where one clock c_i for each task J_i is used to model execution time. In [1] each* TA *has a final state f.*

To treat a JSS we need to compose the automata for each task. This composition takes into account the mutual exclusion principle, by which no more than one task can be active in a machine at any time. The resulting restricted composition is shown in figure 4.13(b).

From the composition automaton, we can derive the different lengths of executions by analysing different runs of the automaton, which represent feasible schedules for $\mathcal{J}$.

Example 4.16 *Two different executions for our previous example are shown bellow, where each tuple is of the form $(m, m', c_1, c_2), m, m' \in \{m_1, \bar{m}_1, m_2, \bar{m}_2\}$ and $\perp$ represents an inactive clock:*

S_1 :

$$(\bar{m}_1, \bar{m}_1, \perp, \perp) \xrightarrow{0} (m_1, \bar{m}_1, 0, \perp) \xrightarrow{4} (m_1, \bar{m}_1, 4, \perp) \xrightarrow{0} (\bar{m}_2, \bar{m}_1, \perp, \perp) \xrightarrow{0}$$
$$(m_2, \bar{m}_1, 0, \perp) \xrightarrow{0} (m_2, m_1, 0, 0) \xrightarrow{3} (m_2, m_1, 3, 3) \xrightarrow{0} (m_2, f, 3, \perp) \xrightarrow{2}$$
$$(m_2, f, 5, \perp) \xrightarrow{0} (f, f, \perp, \perp)$$

S_2 :

$$(\bar{m}_1, \bar{m}_1, \perp, \perp) \xrightarrow{0} (\bar{m}_1, m_1, \perp, 0) \xrightarrow{3} (\bar{m}_1, m_1, \perp, 3) \xrightarrow{0} (\bar{m}_1, f, \perp, \perp) \xrightarrow{0}$$
$$(m_1, f, 0, \perp) \xrightarrow{4} (m_1, f, 4, \perp) \xrightarrow{0} (\bar{m}_2, f, \perp, \perp) \xrightarrow{0} (m_2, f, 0, \perp) \xrightarrow{5}$$
$$(m_2, f, 5, \perp) \xrightarrow{0} (f, f, \perp, \perp)$$

The first schedule S_1 has length 9 while the second S_2 has length 12.

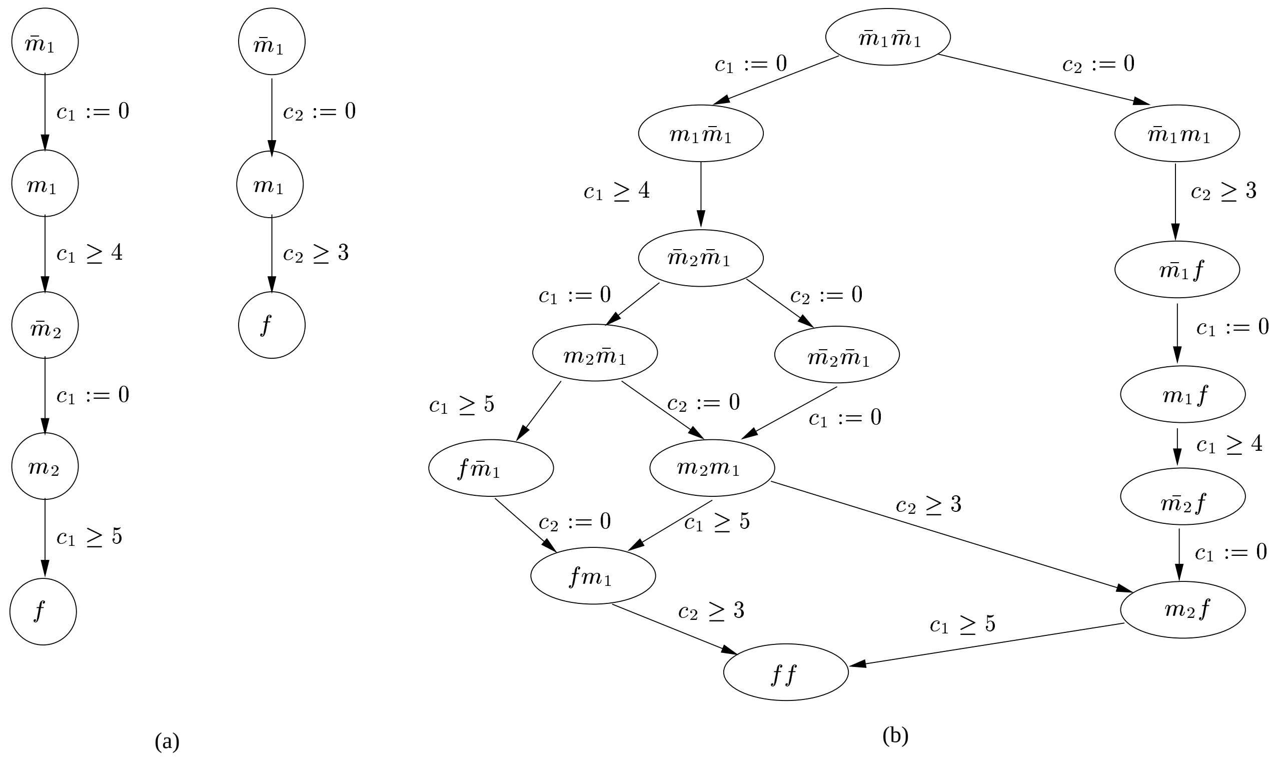


Figure 4.13: Jobs and Timed Automata

The previous example shows the idea for JSS and timed automata, [1]: *the optimal job-shop scheduling problem can be reduced to the problem of finding the shortest path in a acyclic timed automaton.* This problem of reachability, that is arriving to the tuple $(f, f, \perp, \perp)$ is always successful since all runs lead to f. In [1] various techniques for traversing the composed automaton in order to find the shortest path are presented; algorithms reduce the number of explored states, still guaranteeing optimality.

4.8 Conclusions

In this chapter we presented three main streams for modelling and analysing, based on TA and the reachability problem.

Tailored Synthesis The approach studied in [8, 9, 7] is based on the construction of a scheduled system, guided by some desired properties and the application itself. From a TA they construct a new automaton whose invariants must respect the desired properties; priorities are used as inputs and its calculation is guided by conflicting states. The problem of preemption is not clearly handled. Schedulability is attained by construction.

Timed Automata with Tasks The approach studied in [26, 25] is based on the idea of modelling the application under bounded supension automata; the problem of schedulability is reduced to the problem of reachability of an error state; they prove this problem is decidable and hence so is schedulability. The problem with this model is encoding the application in a new UTA, which considers all possible transition among tasks, and then we are soon face to the problem of state explosion. Scheduling policies are fixed priority.

Job Shop The approach studied in [1, 2] is completely different: it treats the problem of (real time) tasks where no deadline restriction is imposed and many machines are at disposition for execution. Modelling is based on the idea of composing the individual models for tasks and schedulability is reduced to a reachability problem with the shortest time weight. Although simple, the problem is too narrow since periodic tasks are not considered (and hence, TA are really acyclic TA) and tasks have no deadlines so, schedulability analysis is almost inexistant.

Chapter 5

The heart of the problem

Résumé

Ce chapitre présente les resultats les plus importants de cette thèse; nous donnons une nouvelle utilisation d'horloges pour modéliser l'ordonnancement dans un cadre avec preemption, dépendences et uncertitude.

Le chapitre présente graduellement notre technique; au debut on considère des systèmes avec une preemption et on les modèlise à l'aide des automates temporisés; on prouve que le problème de l'ordonnancement est decidable en montrant que le problème d'atteignabilité est decidable. On etend notre méthode vers un cadre plus general en presentant une modélisation qui utilise la difference entre deux horloges pour simuler la preemption. Finallement, on conclut par la preuve de decidabilité de cette approche. Un mechanisme d'admission de tâches est presenté basé sur l'idée de temps d'attente.

5.1 Motivation

As seen in the previous chapters, the behavior of real-time systems with preemptive schedulers can be modelled by *stopwatch automata*. Nevertheless, the expressive power of stopwatch automata discouraged for a long time their use for verification purposes. Indeed, the reachability problem (even for a single stopwatch) has been proven to be undecidable [29, 31, 20].

There are, however, some decidable sub-classes such as the so-called *integration graphs* [31] and *suspension automata* [40]. The latter are actually useful for modelling and analyzing systems made up of a set of tasks with fixed execution times. SWA can be translated into timed automata with updates, specifically decrementation by a constant, for which the reachability problem is indeed decidable [18].

The result of [40] has been extended in [26] to a more general model TAT, though still requiring constant execution times of tasks. The approach via timed automata with decrementation suffers of two main problems. First, it requires a costly translation. Second, it only allows modelling tasks with fixed execution times. A technique to cope with the first problem has been proposed in [25].

In this chapter we focus on preemptive scheduling of systems of tasks with uncertain but lower and upper bounded execution times. The behavior of these systems cannot be straightforwardly translated into a decidable extension of timed automata with updates. Our approach consists in encoding the value of stopped clocks as the difference of two running ones. We do allow tasks to be restarted again;

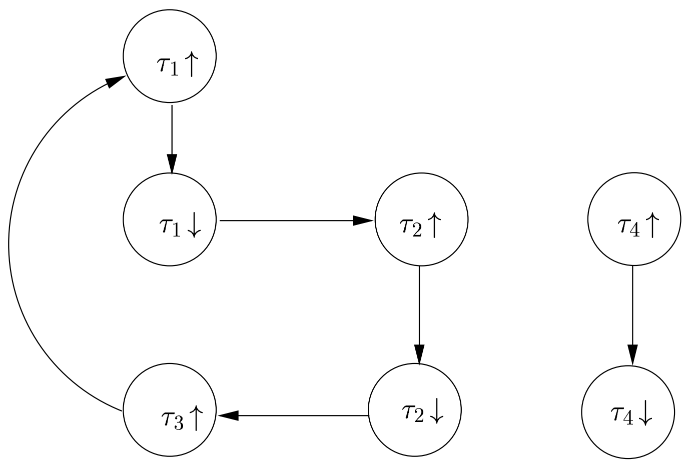


Figure 5.1: A model of a system

initially we forbid preempting a task more than once, then we extend to a more general model. We show that the system can be modelled by formulas involving *difference bounded matrices*, DBM, that is, difference constraints on clocks, and time-invariant equalities capturing the values of stopped clocks. This result implies decidability and leads to an efficient implementation. Moreover, it gives a precise symbolic characterization of the state space for the considered class of systems.

5.2 Model

A real time application is modelled as a collection $\mathbb{T} = \{\tau_1, \tau_2, \ldots, \tau_m\}$ of all tasks for the application, which are triggered by external events, including a timed event such as period. Each task τ_i is characterized by a vector of parameters $[G_i, D_i]$ $1 \leq i \leq m$ where $G_i = [E_i^{\min}, E_i^{\max}]$ is the execution time-interval, i.e. the best and worst case execution time and D_i is the relative deadline. For each task τ_i, we have two timed variables, namely r_i and e_i, that measure the *release* time and the *accumulated executed* time, respectively. Both variables are reset to zero whenever task τ_i arrives. Task arrival is denoted by $\tau_i\uparrow$ and task completion by $\tau_i\downarrow$.

The *environment* is any untimed relation between arrivals and completions of all tasks (it should respect the precedence relationship between $\tau_i\uparrow$ and $\tau_i\downarrow$, though).

Example 5.1 *In figure 5.1 we can see an example of a model of system of four tasks.*

The general model of an application is then a graph $\mathcal{G} = (V, A)$, where the set of vertices $V \subseteq \{\tau_i\uparrow, \tau_i\downarrow\}_{1\leq i\leq m}$ where $|\mathbf{in}(V)| \leq 1$ (no more than one incoming edge per vertex) and a set of directed edges

$$A \subseteq \{\tau_i\uparrow \rightarrow \tau_i\downarrow\}_{1\leq i\leq m} \cup \{\tau_i\downarrow \rightarrow \tau_j\uparrow\}_{i\neq j,\ 1\leq i,j\leq m} \cup \{\tau_i\uparrow \rightarrow \tau_j\uparrow\}_{i\neq j,\ 1\leq i,j\leq m}$$

Let $T \subseteq \mathbb{T}$ be the finite set of *active* tasks in the system, that is, those that have already arrived and are currently being handled by the scheduler. At any moment, at most one instance of a task may be active. The predicate $\mathbf{exec}(T_i)$ indicates whether T_i is executing or not and for the set T, $\mathbf{exec}(T)$ denotes the executing task of T. The predicate $\mathbf{accept}(\tau_i)$ indicates whether τ_i is accepted or not at its arrival due to some scheduling or modelling reasons; for instance, τ_i could be rejected because it would produce some tasks to miss their deadlines or because there is an active instance of this task. A detailed description of this predicate will be given when we precise the scheduling policy. Figure 5.2 shows a task automaton.

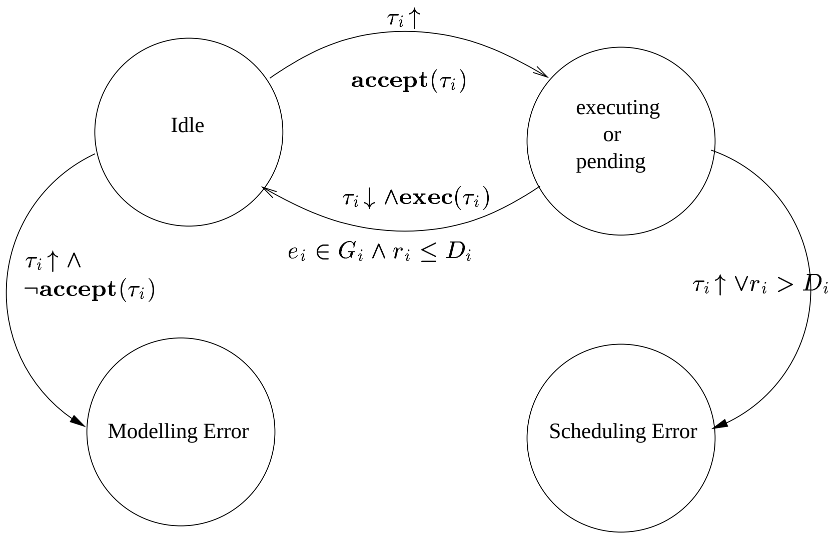

$\tau_i\uparrow$
$\mathbf{accept}(\tau_i)$
Idle
executing or pending
$\tau_i\downarrow \wedge \mathbf{exec}(\tau_i)$
$e_i \in G_i \wedge r_i \leq D_i$
$\tau_i\uparrow \wedge$ $\neg\mathbf{accept}(\tau_i)$
$\tau_i\uparrow \vee r_i > D_i$
Modelling Error
Scheduling Error

Figure 5.2: Task automaton

The dynamic behavior of the system is represented by a transition system $(\mathcal{S}, T, \rightarrow)$ where $\mathcal{S}$ is a set of states, T is the set of active tasks, and $\rightarrow$ is the transition relation. $\mathcal{S}$ is a tuple of control locations of the task automaton (Fig. 5.2) and of valuations of timed variables. The following rules give a sketched behaviour of the system; formal and complete rules will be given later when analysing particular scheduling policies, SCH.

- **Task completion**: $\tau_i\downarrow$ If $e_i \in G_i$, $r_i \leq D_i$, and $\mathbf{exec}(\tau_i)$, then

$$(\mathcal{S}, T) \overset{\tau_i\downarrow}{\rightarrow} (\mathcal{S}', T \ominus \{\tau_i\})$$

where $\mathcal{S}'$ is obtained according to SCH and $\ominus$ is the operation of removing a task from T.

- **Task arrival**: $\tau_i\uparrow$
If $\tau_i \in T \Rightarrow$

$$(\mathcal{S}, T) \overset{\tau_i\uparrow}{\rightarrow} \text{Schedulling Error}$$

(no more than one instance of each task)
If $\tau_i \notin T \wedge \neg\mathbf{accept}(\mathcal{S}, T, \tau_i) \Rightarrow$

$$(\mathcal{S}, T) \overset{\tau_i\uparrow}{\rightarrow} \text{Modelling Error}$$

If $\mathbf{accept}(\mathcal{S}, T, \tau_i) \Rightarrow$

$$(\mathcal{S}, T) \overset{\tau_i\uparrow}{\rightarrow} (\mathcal{S}', T \oplus \{\tau_i\})$$

where S' is obtained according to SCH and $\oplus$ is the operation of inserting a task in T.

- **Deadline violation**: If $r_i > D_i$ for some $\tau_i \in T \Rightarrow$

$$(\mathcal{S}, T) \to \text{Schedulling Error}$$

- **Time passing**: Let $\mathbf{exec}(\tau_i)$, $\delta \geq 0$, and $e_i + \delta \leq E_i^{\max} \Rightarrow$

$$(\mathcal{S}, T) \xrightarrow{\delta} (\mathcal{S}', T)$$

where S' is obtained from S by adjusting the values of timed variables according to SCH.

The first rule expresses the completion of the executing task, leaving to the scheduler the choice of choosing the next task to be executed. The definition of $\mathbf{exec}(T)$ and the computation of the next state are left unspecified since they are dependant of the SCH.

The second rule expresses the arrival of a new task, which can be accepted or rejected. We distinguish two transitions leading to an error state, one for unschedulability and the other for a behaviour not satisfying the modelling assumptions.

The third rule expresses the case of a deadline violation. The fourth rule expresses time passing of δ units of time, adjusting the values of the timed variables in T.

In general we will assume the existence of an acceptance test at task arrival. This test is related to some assumptions of our system and of course, to the scheduling policy. Once the task passes the test, it can enter the system, either waiting for its turn or executing immediately, preempting the currently executing task. Predicate $\mathbf{accept}(\mathcal{S}, T, \tau_i)$ will be analysed in detail for different scheduling policies.

5.3 LIFO scheduling

To show our analysis, we start with a very simple scheduling policy: a LIFO scheduler, that is, a scheduler where the current executing task is always preempted by the recently arrived task. We also suppose that each task can be preempted for at most once.

Intuitively speaking, a one preemption LIFO scheduler accepts tasks in the stack, until the task on the top finishes; at this moment, as all tasks beneath it had already been preempted, the scheduler will reject any new task, until the stack is empty; note, then, that all tasks in the stack had been preempted once, except the task on the top which could have never been preempted.

Example 5.2 *Let $\mathbb{T} = \{\tau_1(4,12), \tau_2(5,10), \tau_3(2,10), \tau_4(3,6)\}$ be a set of tasks, where the numbers in parentheses represent execution times and deadlines. In this example, deadlines are sufficiently long to let all tasks execute on time.*

Figure 5.3 shows the reaction of a LIFO scheduler at arrival of each task. For instance at time $t = 3$, $\tau_2 \uparrow$ and $\tau_2 \dagger \tau_1$; note that at time $t = 8$, $\tau_3 \downarrow$, and τ_2 resumes execution, noted as $\tau_2 \nearrow$; remark that the arrival of τ_4 at time $t = 9$ is ignored by the scheduler, since τ_2 had already been preempted. We also show the evolution of clocks.

5.3.1 LIFO Transition Model

Let $T = \{T_1, T_2, \ldots T_n\}$ be the *stack* of active tasks in the system and let T_n be the task in execution, i.e. $T_n = \mathbf{exec}(T)$; if τ_i arrives to the system and it is accepted, then τ_i preempts T_n, written as $\tau_i \dagger T_n$.

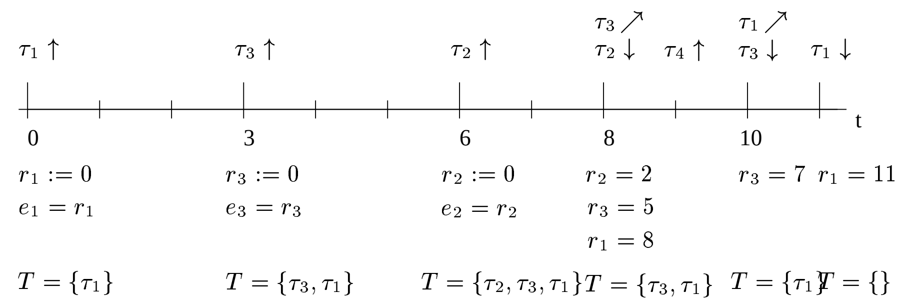


Figure 5.3: One preemption LIFO Scheduler

We define a function μ for renaming tasks, $\mu : T \to \{1, 2, \ldots, m\}$, $\mu(T_i) = j$, $1 \leq i \leq n$, $1 \leq j \leq m$ gives the "name" j in $\mathbb{T}$ of the task τ_j placed in position i in stack T.

An hybrid transition system $(\mathcal{S}, T, \rightarrow)$ for a LIFO scheduler is composed of :

1. a collection of states, $\mathcal{S} = (S, \vec{e}, \vec{r}, \vec{\dot{e}}, \vec{p})$, where:
   (a) S is a control location,
   (b) $\vec{e}$ a vector of clocks counting execution time, where $\vec{e}_j$ is the cumulated execution time for task τ_j,
   (c) $\vec{r}$ a vector for releasing times, where $\vec{r}_j$ is the released time for task τ_j,
   (d) $\vec{\dot{e}}$ a vector indicating those execution clocks which are stopped,
   (e) $\vec{p}$ a vector for preemption where $\vec{p}_j = l$ means that task τ_l preempts τ_j, $\tau_l \dagger \tau_j$, if $l \neq 0$ or that τ_j has never been preempted, otherwise. In particular, in the LIFO scheduler, if task τ_j is at position $k, 1 \leq k < n$ in T, that is $\tau_j = T_k$, then task τ_l is at position $k+1$, that is[1] :

$$\mu(T_k) = j,\ \mu(T_{k+1}) = l,\ \vec{p}_{\mu(T_k)} = \mu(T_{k+1}) \equiv \vec{p}_j = l$$

2. a stack of tasks, $T \subseteq \mathbb{T}$ (with the usual operations **pop**, **top** and **push**).
3. a transition relation $\rightarrow$

We give the operations over our transition system; recall that the arrival of a task is captured by the scheduler, who decides over the admission. As a convention, we will use index j for "names" of tasks, $1 \leq j \leq m$ and k for active tasks in T, $1 \leq k \leq n$.

- Task arrival, $\tau_i\uparrow$ ($\tau_i \in \mathbb{T}$ and $\textbf{accept}(\mathcal{S}, T, \tau_i)$):

$$(\mathcal{S}, T) \xrightarrow{\tau_i\uparrow} (\mathcal{S}', T')$$

  where $T' \equiv \textbf{push}(\tau_i, T)$ and $\mathcal{S}' = (S', \vec{e'}, \vec{r'}, \vec{\dot{e}'}, \vec{p'})$ is:

$$\vec{e'}_j = \begin{cases} 0 & \text{if } i = j \\ \vec{e}_j & \text{otherwise} \end{cases}$$

[1]We could simplify $\vec{p}$ as a boolean vector where $\vec{p}_j = \theta$, and $\theta \in [\text{true}, \text{false}]$

$$\vec{r'}_j = \begin{cases} 0 & \text{if } i = j \\ \vec{r}_j & \text{otherwise} \end{cases}$$

$$\vec{p'}_j = \begin{cases} i & \text{if } j = \mu(\mathbf{top}(T)) \\ 0 & \text{if } j = i \\ \vec{p}_j & \text{otherwise} \end{cases}$$

$$\vec{\dot{e}}'_j = \begin{cases} 1 & \text{if } j = i \\ 0 & \text{otherwise} \end{cases}$$

That is, as a new task is accepted, its execution and release clocks are both reset, its rate execution clock is set to 1 to mark it is running, while all other execution clocks are stopped; we mark preemption to the current executing task.

- Task completion: $T_n\!\downarrow$

$$(\mathcal{S}, T) \xrightarrow{T_n\downarrow} (\mathcal{S}', T')$$

where $T' = \mathbf{pop}(T)$ and $\vec{e}_{\mu(T_n)} = \vec{r}_{\mu(T_n)} = \perp$, all other variables are unchanged.

- Task resumption: $T_n\!\nearrow$ (we assume $T_n = \text{top}(T)$ is a task preempted in the past, which regains the processor).

$$(\mathcal{S}, T) \xrightarrow{T_n\nearrow} (\mathcal{S}', T)$$

where $\mathcal{S}' = (S', \vec{e'}, \vec{r'}, \vec{\dot{e}'}, \vec{p'})$ is:

$$\vec{\dot{e}}'_j = \begin{cases} 1 & \text{if } j = \mu(T_n) \\ 0 & \text{otherwise} \end{cases}$$

all other variables remain unchanged.

- Time passing: δ is an elapsed time not enough to finish the current executing task.

$$(\mathcal{S}, T) \xrightarrow{\delta} (\mathcal{S}', T)$$

where $\mathcal{S}' = (S', \vec{e'}, \vec{r'}, \vec{\dot{e}'}, \vec{p'})$ is:

$$\vec{e'}_j = \begin{cases} \vec{e}_j + \delta & \text{if } j = \mu(T_n) \\ \vec{e}_j & \text{otherwise} \end{cases}$$

and $\vec{r'}_j = \vec{r}_j + \delta \; \forall \tau_j, \tau_j \in T$, all other variables remain unchanged.

5.3.2 LIFO Admittance Test

It is time to give an admittance test for our LIFO scheduler. We propose:

$$\textbf{accept}_{\text{LIFO}}(\mathcal{S}, T, \tau_i) \equiv \neg\textbf{active}(T, \tau_i) \wedge \neg\textbf{preempted}(T, T_n)$$

that is, we do not accept a task if:

- There is an active instance of the same task, that is

$$\textbf{active}(T, \tau_i) \equiv i = \mu(T_k) \text{ for some } k, 1 \leq k \leq n$$

- It will preempt an already preempted task, that is

$$\textbf{preempted}(T, T_n) \equiv \vec{p}_{\mu(T_n)} \neq 0$$

For the instant being we do not consider timing constraints; in particular, we are not considering in the acceptance test the fact that a new accepted task may lead to some other tasks in T miss their deadlines. Later, we propose a refinement in that direction.

5.3.3 Properties of LIFO scheduler

Under the one-preemption assumption the following properties hold:

1. If $\vec{p}_{\mu(T_n)} = 0 \Rightarrow e_{\mu(T_n)} = r_{\mu(T_n)}$
2. $e_{\mu(T_{n-1})} = r_{\mu(T_{n-1})} - r_{\mu(T_n)}$
3. $\forall T_k, T_k \in T, k < n$, we have:
   (a) Preemption: $\textbf{active}(T_k) \wedge \textbf{preempted}(T_k)$
   (b) Time invariant condition:

$$I_{\text{LIFO}}(T) \equiv \bigwedge_{k=1}^{n-1} e_{\mu(T_k)} = r_{\mu(T_k)} - r_{\mu(T_{k+1})}$$

$$I_{\text{LIFO}}(T) \equiv \bigwedge_{k=1}^{n-1} e_{\mu(T_k)} = r_{\mu(T_k)} - r_{\vec{p}_{\mu(T_k)}}$$

   (c) Schedulability: $r_{\mu(T_k)} < D_{\mu(T_k)}$

Property 1 simply says that if the currently executing task T_n has never been preempted since its arrival, then both clocks, $e_{\mu(T_n)}$ and $r_{\mu(T_n)}$ have the same value.

Property 2 is the consequence of preemption. When T_{n-1} was preempted, (i.e. T_{n-1} was executing and hence on top of T), we know by the previous rule, that $e_{\mu(T_{n-1})} = r_{\mu(T_{n-1})}$ and as $r_{\mu(T_n)}$ is set to zero, we can establish the property which is time-invariant while T_{n-1} is suspended. This observation leads by induction to property 3b, which we call the *execution invariant* under a LIFO scheduling policy.

Property 3a says that all tasks in stack T are active and were preempted in the past (except eventually the task in the top).

Property 3c says that all tasks in T are schedulable, (remember our extension of the admittance test will go in that direction).

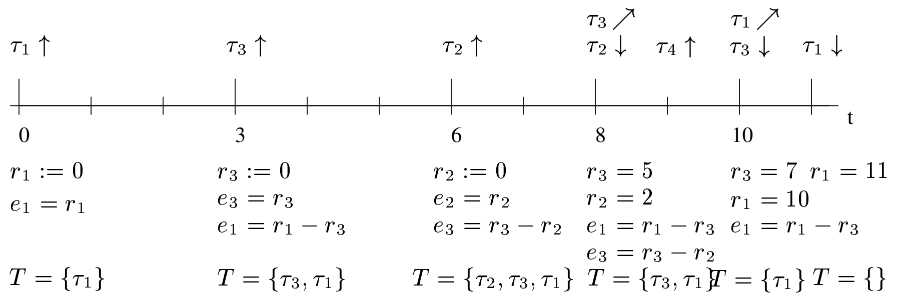


Figure 5.4: Invariants in LIFO Scheduler

Example 5.3 *Let us reconsider the example 5.2; we can observe that (see figure 5.4):*

- At $t = 0$ $r_1 = e_1 = 0$, τ_1 begins its execution. $I_{\text{LIFO}}(\tau_1) = true$,
- At $t = 3$, τ_3 arrives, it is accepted and preempts τ_1 (later, we will give an admission test dealing with schedulability conditions). The execution invariant $I_{\text{LIFO}}(\tau_1, \tau_2) \equiv \{e_1 = r_1 - r_3\}$
- At $t = 6$, $\tau_2 \uparrow$ and $\tau_2 \dagger \tau_3$. The execution invariant $I_{\text{LIFO}}(\tau_1, \tau_2, \tau_3) \equiv \{e_1 = r_1 - r_3 \wedge e_3 = r_3 - r_2\}$.
- At $t = 8$, τ_2 completes its execution and the scheduler resumes τ_3. The computed time is recovered from the difference $e_3 := r_3 - r_2$. Note that $\vec{p}_3 = 2$ and $e_3 = r_3 - r_{\vec{p}_3}$.
- At $t = 9$, T_4 arrives but it is rejected by the admittance test since: $\mathbf{exec}(T) = \tau_3 \wedge \mathbf{preempted}(\tau_3)$.
- Finally, at $t = 10$, τ_3 completes and the scheduler resumes τ_1; once again e_1 is recovered from the difference $r_1 - r_3$; τ_1 ends at $t = 11$.

The previous properties motivate the following definition:

Definition 5.1 *The* **execution invariant** under a LIFO scheduling policy is

$$I_{\text{LIFO}}(T) = \bigwedge_{1 \leq k \leq n-1} e_{\mu(T_k)} = r_{\mu(T_k)} - r_{\mu(T_{k+1})}$$

If the currently executing task has already been preempted, the equation $r_{\mu(T_n)} = e_{\mu(T_n)}$ may not hold and in this case, we cannot simply express $e_{\mu(T_n)}$ as the difference of $r_{\mu(T_n)}$ and $r_{\mu(T_{n+1})}$. So for the time being, we still retain our assumption of one-preemption. To ensure that $r_{\mu(T_n)} = e_{\mu(T_n)}$ holds, we can constrain the predicate $\mathbf{accept}(\mathcal{S}, T, \tau_i)$ for every task τ_i by the constraint

$$e_{\text{exec}(T)} = r_{\text{exec}(T)}$$

5.3.4 Reachability Analysis in LIFO Scheduler

Let Ψ be the set of formulas generated by the following grammar:

$$\psi ::= x - y \leq d \mid \psi \wedge \psi \mid \neg\psi \mid \exists x.\psi$$

where $x, y \in \mathcal{C}$ are clocks and $d \in \mathbb{Q}$ is a rational constant.

To facilitate notation, we will skip in this analysis the use of the function μ, and replace it by the position a task occupies in the stack. Remember then that when saying, for instance, e_k we really mean the execution clock e of the task which is in the k position in the stack, that is $e_{\mu(T_k)}$.

Let ϕ be a constraint characterizing a set of states. We define $T_{n+1}\uparrow(\phi)$ to be the set of states reached when task T_{n+1} arrives, that is:

$$T_{n+1}\uparrow(\phi) = \{s' : \exists s \in \phi.\ s \overset{T_{n+1}\uparrow}{\to} s'\}$$

Let ϕ be of the form $I_{\text{LIFO}}(T) \wedge \psi$, with $\psi \in \Psi$ a quantifier free formula. Without loss of generality, we can assume that either :

1. T_{n+1} is rejected:
$$\psi \implies \neg\textbf{accept}(\psi, T, T_{n+1})$$
in this case, $T_{n+1}\uparrow(\phi) \equiv \phi$ and the system moves to an error state.

2. T_{n+1} is accepted:
$$\psi \implies \textbf{accept}(\psi, T, T_{n+1})$$
We have that:
$$T_{n+1}\uparrow(\phi) \equiv I_{\text{LIFO}}(T \oplus \{T_{n+1}\}) \wedge e_{n+1} = r_{n+1} = 0 \wedge \exists e_n.\psi$$
Moreover, since $e_n = r_n$, $I_{\text{LIFO}}(T \oplus \{T_{n+1}\})$ contains the equality $e_n = r_n - r_{n+1}$, we have that:
$$T_{n+1}\uparrow(\phi) \equiv I_{\text{LIFO}}(T + \{T_{n+1}\}) \wedge e_{n+1} = r_{n+1} = 0 \wedge \psi'$$
Hence, $T_{n+1}\uparrow(\phi)$ has the same structure than ϕ, that is, it is the conjunction of an execution invariant and a formula in Ψ. Moreover, if ψ is a quantifier-free formula, that is, a difference constraint (or DBM), we have that $\exists e_n.\psi$ is indeed a DBM. Note that ψ is a formula containing clocks measuring release times and only *one* execution clock (that of the task on top).

Then we have:

Proposition 5.1 *Let ϕ be of the form $I_{\text{LIFO}} \wedge M$, where M is a* DBM *and I_{LIFO} is a one preemption* LIFO *execution invariant, then, $T_{n+1}\uparrow(\phi)$ has the same structure as ϕ.*

Now, let $\phi\nearrow$ the set of states reached from ϕ by letting time advance, that is:

$$\phi\nearrow = \{s' : \exists s \in \phi, \delta \geq 0. s \xrightarrow{\delta} s'\}$$

Clearly, if ϕ is of the form $I_{\text{LIFO}}(T) \wedge M$, we have that

$$\phi\nearrow = (I_{\text{LIFO}}(T) \wedge M)\nearrow = I_{\text{LIFO}} \wedge M\nearrow$$

Proposition 5.2 *Let ϕ be of the form $I_{\text{LIFO}} \wedge M$, where M is a* DBM *and I_{LIFO} is a one preemption* LIFO *execution invariant, then, $\phi\nearrow$ has the same structure as ϕ.*

Thus, given a sequence of task arrivals $T_1\uparrow, \ldots, T_n\uparrow$, the set of reached states can be represented by the conjunction of the execution invariant $I_{\mathrm{LIFO}}(T)$, characterizing the already executed time of the suspended tasks, namely $T_1, \ldots, T_{n-1}$, and a DBM M, characterizing the relationship between the corresponding released times and the equality $e_n = r_n$.

A DBM M has the following form:

$$\begin{array}{llllll}
 & u & r_1 & r_2 & \ldots & r_n & e_n \\
u & - & M_{ur_1} & M_{ur_2} & \ldots & M_{ur_n} & M_{ue_n} \\
r_1 & M_{r_1u} & - & M_{r_1r_2} & \ldots & M_{r_1r_n} & M_{r_1e_n} \\
r_2 & M_{r_2u} & M_{r_2r_1} & - & \ldots & M_{r_2r_n} & M_{r_2e_n} \\
\vdots & & & & & & \\
r_n & M_{r_nu} & M_{r_nr_1} & M_{r_nr_2} & \ldots & - & M_{r_ne_n} \\
e_n & M_{e_nu} & M_{e_nr_1} & M_{e_nr_2} & \ldots & M_{e_nr_n} & -
\end{array}$$

As $e_n = r_n$, then $M_{xe_n} = M_{xr_n}$ and $M_{e_nx} = M_{r_nx}$, so from here on we omit e_n in M.

In our case, M is constructed in a very particular way and therefore has a special structure. Let us analyse it:

Event	*Equation*	*Explanation*	
$T_1\uparrow$	$\emptyset$		
$T_2\uparrow$	$M_{r_1r_2} = M_{r_1u}$	$\blacklozenge\ e_1 = r_1 - r_2 \leq M_{r_1r_2} = M_{r_1u} \because r_2 = u$	(5.1)
$T_3\uparrow$	$M_{r_2r_3} = M_{r_2u}$	$\blacklozenge\ e_2 = r_2 - r_3 \leq M_{r_2r_3} = M_{r_2u} \because r_3 = u$	(5.2)
	$M_{r_1r_3} = M_{r_1r_2} + M_{r_2r_3}$	$(e_1 = r_1 - r_2 \leq M_{r_1r_2} \wedge e_2 = r_2 - r_3 \leq M_{r_2r_3}) \Rightarrow$	
		$r_1 - r_3 \leq M_{r_1r_3} = M_{r_1r_2} + M_{r_2r_3}$	(5.3)
$T_4\uparrow$	$M_{r_3r_4} = M_{r_3u}$	$\blacklozenge\ e_3 = r_3 - r_4 \leq M_{r_3r_4} = M_{r_3u} \because r_4 = u$	(5.4)
	$M_{r_2r_4} = M_{r_2r_3} + M_{r_3r_4}$	$(e_2 = r_2 - r_3 \leq M_{r_2r_3} \wedge e_3 = r_3 - r_4 \leq M_{r_3r_4}) \Rightarrow$	
		$r_2 - r_4 \leq M_{r_2r_4} = M_{r_2r_3} + M_{r_3r_4}$	(5.5)
	$M_{r_1r_4} = M_{r_1r_2} + M_{r_2r_3} + M_{r_3r_4}$	$(e_1 = r_1 - r_2 \leq M_{r_1r_2} \wedge e_2 = r_2 - r_3 \leq M_{r_2r_3}$	
		$\wedge e_3 = r_3 - r_4 \leq M_{r_3r_4}) \Rightarrow$	
		$r_1 - r_4 \leq M_{r_1r_4} = M_{r_1r_2} + M_{r_2r_3} + M_{r_3r_4}$	(5.6)

We can observe that equality 5.3 is deduced from 5.1 and 5.2; 5.5 from 5.2 and 5.4 and finally 5.6 from 5.1, 5.2 and 5.4. So we see that the matrix is constructed from a set of **base formulae** corresponding to the difference of the task being executed and that which preempts it, while all other differences can be constructed from this base set. Base formulae are marked with a $\blacklozenge$.

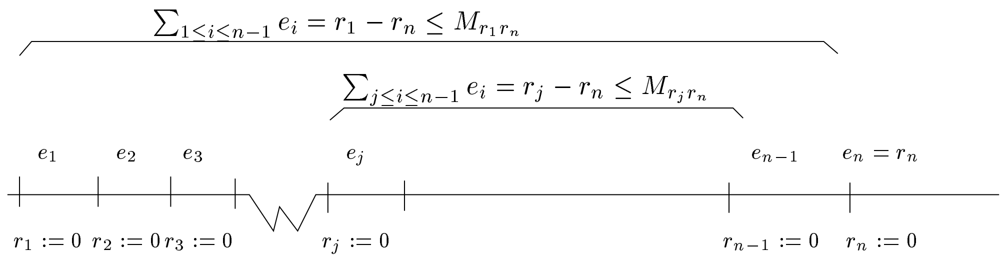
In general, when T_n arrives:

$$M_{r_{n-1}r_n} = M_{r_{n-1}u}$$

and

$$M_{r_{n-1}e_{n-1}} = M_{e_{n-1}r_{n-1}} = 0$$

As T_n preempts T_{n-1} and



$\sum_{1<i<n-1} e_i = r_1 - r_n \leq M_{r_1 r_n}$
$\sum_{j\leq i\leq n-1} e_i = r_j - r_n \leq M_{r_j r_n}$
e_1
e_2
e_3
e_j
e_{n-1}
$e_n = r_n$
$r_1 := 0$
$r_2 := 0$
$r_3 := 0$
$r_j := 0$
$r_{n-1} := 0$
$r_n := 0$

Figure 5.5: Clock Differences in LIFO Scheduler

$$M_{r_j r_n} = \sum_{i=j}^{n-1} M_{r_i r_{i+1}},\ j < n-1 \tag{5.7}$$

Equation 5.7 may be re-written as a recursive formula:

$$M_{r_j r_n} = M_{r_j r_{n-1}} + M_{r_{n-1} r_n},\ j < n-1 \tag{5.8}$$

and

$$M_{r_j u} = M_{r_j r_n} + M_{r_n u}$$

What do these differences mean? Figure 5.5 shows a geometric interpretation of the following relations:

$$\begin{array}{cccc} e_1 = & r_1 - r_2 & \leq & M_{r_1 r_2} \\ e_2 = & r_2 - r_3 & \leq & M_{r_2 r_3} \\ \vdots\ \ \vdots & \vdots & \vdots & \\ e_{n-1} = & r_{n-1} - r_n & \leq & M_{r_{n-1} r_n} \\ \hline \sum_{1\leq i<n} e_i = & \sum_{1\leq i<n} r_i - r_{i+1} & \leq & \sum_{1\leq i<n} M_{r_i r_{i+1}} \end{array}$$

$$\sum_{1\leq i<n} e_i = r_1 - r_n \leq \sum_{1\leq i<n} M_{r_i r_{i+1}} \tag{5.9}$$

On the other hand, we know that:

$$r_1 - r_n \leq M_{r_1 r_n} \tag{5.10}$$

From the expression (5.9) and (5.10) and (5.7) we can deduce that:

$$\sum_{1\leq i<n} e_i = r_1 - r_n \leq \sum_{1\leq i<n} M_{r_i r_{i+1}} = M_{r_1 r_n}$$

The expression 5.9 can be generalized as:

$$\sum_{j\leq i<n} e_i = r_j - r_n \leq M_{r_j r_n}$$

That is, when T_n arrives we have that:

$$\begin{aligned} M_{r_{n-1}r_n} &= M_{r_{n-1}u} && (5.11)\\ M_{r_{n-1}e_{n-1}} &= M_{e_{n-1}r_{n-1}} = 0 && (5.12)\end{aligned}$$

and for all $j < k \leq n$,

$$\begin{aligned} M_{r_j u} &= M_{r_j r_k} + M_{r_k u} && (5.13)\\ M_{u r_j} &= M_{u r_k} + M_{r_k r_j} && (5.14)\\ M_{r_j r_k} &= M_{r_j r_{k-1}} + M_{r_{k-1} r_k} && (5.15)\\ M_{r_k r_j} &= M_{r_k r_{k-1}} + M_{r_{k-1} r_j} && (5.16)\end{aligned}$$

Let us call a DBM M that satisfies properties 5.11 to 5.16 a **nice** DBM.

We have therefore proved that:

Proposition 5.3 *$T_{n+1}\uparrow(\cdot)$ and $\nearrow(\cdot)$ preserve nicety.*

We have shown so far that task arrival and time passing preserve the structure of the symbolic characterization of the state space for a LIFO scheduler if tasks are accepted only under the one-preemption restriction. The question that arises then, is whether task completion has the same property. If this is the case, we have a complete symbolic characterization of the state space of such schedulers. Indeed, the answer is yes, though the reasoning is a bit more involved.

Let $T_n\downarrow(\phi)$ be the set of states reachable from ϕ when T_n terminates:

$$T_n\downarrow(\phi) = \{s' : \exists s \in \phi . s \overset{T_n\downarrow}{\rightarrow} s'\}$$

Let ϕ be of the form $I_{\text{LIFO}}(T) \wedge M$ and G_n be the interval $[E_n^{\min}, E_n^{\max}]$. We have that:

$$\begin{aligned} T_n\downarrow(\phi) &\equiv \exists e_n, r_n . I_{\text{LIFO}}(T) \wedge M \wedge G_n\\ &\equiv I_{\text{LIFO}}(T - \{T_n\}) \wedge \exists r_n . e_{n-1} = r_{n-1} - r_n \wedge \exists e_n . (M \wedge G_n)\\ &\equiv I_{\text{LIFO}}(T - \{T_n\}) \wedge M'[r_n \leftarrow r_{n-1} - e_{n-1}]\end{aligned}$$

where $M' \equiv \exists e_n . (M \wedge G_n)$, that is we eliminate e_n from M, since we do not need it. The question is: " Is M' still a nice DBM matrix?"

We have to show now that $M'[r_n \leftarrow r_{n-1} - e_{n-1}]$ is equivalent to a nice DBM.

When substituting r_n by $r_{n-1} - e_{n-1}$ in M we get:

$$\text{from } r_n - r_{n-1} \leq M_{r_n r_{n-1}} \Rightarrow -e_{n-1} \leq M_{r_n r_{n-1}} \qquad (5.17)$$

$$\text{from } r_n - r_j \leq M_{r_n r_j} \Rightarrow r_{n-1} - e_{n-1} - r_j \leq M_{r_n r_j} \qquad (5.18)$$

which is not a difference constraint, but from (5.17) and

$$r_{n-1} - r_j \leq M_{r_{n-1} r_j}$$

we derive that

$$r_{n-1} - e_{n-1} - r_j \leq M_{r_n r_{n-1}} + M_{r_{n-1} r_j}$$

Since M is nice, we have that:

$$M_{r_n r_j} = M_{r_n r_{n-1}} + M_{r_{n-1} r_j}$$

which means that (5.18) is an implied constraint and it can be eliminated. The same applies for all the non-difference constraints that appear after subsitution. Since no other new constraints on released time variables appear, nicety is preserved.

In summary:

Proposition 5.4 *Let ϕ be of the form $I \wedge M$, where M is a nice* DBM *and I is a* LIFO *execution invariant. Then, $T_n \downarrow (\phi)$ has the same structure than ϕ.*

Since all variables are bounded, the above results imply the following:

Theorem 5.1 *The symbolic reachability graph of a system of tasks for a* LIFO *scheduler satisfying the one-preemption constraint is finite.*

Hence, the reachability (and therefore the schedulability) problem for our class of systems is decidable. More importantly, our result gives a fully symbolic characterization of the reach-set.

5.3.5 Refinement of LIFO Admittance Test

We have "skipped" the analysis of deadlines; in this section we give a refinement of our LIFO admittance test.

We propose to test at $\tau_i \uparrow$ the predicate:

$$\mathbf{accept}_{\text{LIFO}}(T, \tau_i) \equiv \neg\mathbf{active}(T, \tau_i) \wedge \neg\mathbf{preempted}(T, T_n) \wedge \mathbf{schedulable}(T, \tau_i)$$

that is, we do not accept a task if:

- There is an active instance of the same task, that is $\mathbf{active}(T, \tau_i) \equiv i = \mu(T_k)$ for some $k, 1 \leq k \leq n$.

- It will preempt an already preempted task, that is $\mathbf{preempted}(T, T_n) \equiv \vec{p}_{\mu(T_n)} \neq 0$.

- It will miss its deadline or cause other tasks in T miss their deadlines, i.e,

$$\mathbf{schedulable}(T, \tau_i) \equiv \forall \tau_j \in \{T \cup \tau_i\},\ r_j + B_j + (E_j^M - e_j) \leq D_j \tag{5.19}$$

  that is we must calculate how many units of time the r_j's will be shifted after computing those tasks which have higher priorities, including τ_i, and of course how many units of time must at most execute τ_j. The time a task τ_j will be suspended as a consequence of the execution of higher priority tasks is called the *blocking time*, B_j.

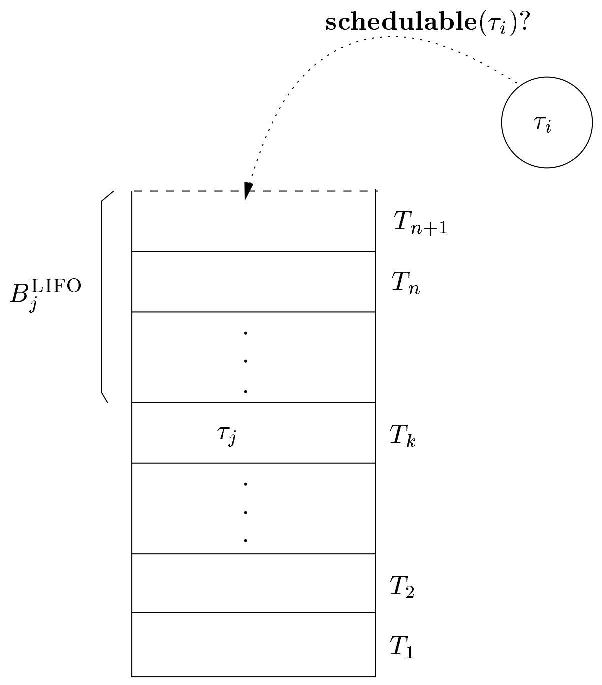


Figure 5.6: Tasks in a LIFO scheduler

We should note that a LIFO scheduler is in some kind a dynamic priority protocol, since the arrival of a new task will, in principle, preempt the currently executing one. That is, priorities are given by task arrival and hence by stack position, $\pi_{\mu(T_1)} < \pi_{\mu(T_2)} \cdots < \pi_{\mu(T_n)}$.

At task arrival, the schedulability test must assure no deadline missing for all active tasks, which, due to the one-preemption hypothesis it does not necessarily mean that the new task will be accepted.

For each task $T_k \in T$, the blocking time when a new task τ_i arrives can be calculated as follows $(j = \mu(T_k), 1 \leq k \leq n0)$:

$$B_j^{\text{LIFO}} = \sum_{\pi_{\tau'} > \pi_j} (E_{\tau'} - e_{\tau'})$$

under the LIFO scheduler we know that $e_j = r_j - r_{\vec{p}_j}\ \forall\, k < n, T_k \in T, \mu(T_k) = j$. Obviously $e_{\mu(T_n)} = r_{\mu(T_n)}$ (if not, τ_i would violate the one preemption hypothesis and hence it should not be accepted) and $e_{\mu(T_{n+1})} = 0$, so (see figure 5.6 for a graphical interpretation of these formula):

$$\begin{aligned} B_j^{\text{LIFO}} &= \sum_{l=k+1}^{n+1} (E^M_{\mu(T_l)} - e_{\mu(T_l)}) \\ B_j^{\text{LIFO}} &= \sum_{l=k+1}^{n-1} (E^M_{\mu(T_l)} - e_{\mu(T_l)}) + (E^M_{\mu(T_n)} - e_{\mu(T_n)}) + E^M_{\mu(T_{n+1})} \end{aligned}$$

Using our execution time invariant and the fact that $\tau_{\mu(T_n)}$ cannot have been preempted (if this were the case, then surely τ_i cannot be accepted), we have:

$$\begin{aligned}
B_j^{\mathrm{LIFO}} &= \sum_{l=k+1}^{n-1} (E^M_{\mu(T_l)} - (r_{\mu(T_l)} - r_{\mu(T_{l+1})})) + (E^M_{\mu(T_n)} - r_{\mu(T_n)}) + E^M_{\mu(T_{n+1})} \\
B_j^{\mathrm{LIFO}} &= \sum_{l=k+1}^{n+1} (E^M_{\mu(T_l)}) - \sum_{l=k+1}^{n-1} (r_{\mu(T_l)} - r_{\mu(T_{l+1})}) - r_{\mu(T_n)} \\
B_j^{\mathrm{LIFO}} &= \sum_{l=k+1}^{n+1} (E^M_{\mu(T_l)}) - (r_{\mu(T_{k+1})} - r_{\mu(T_n)}) - r_{\mu(T_n)} \\
B_{\mu(T_k)}^{\mathrm{LIFO}} &= \sum_{l=k+1}^{n+1} (E^M_{\mu(T_l)}) - r_{\mu(T_{k+1})} \qquad (5.20)
\end{aligned}$$

Replacing in 5.19 with 5.20, using $j = \mu(T_k)$ we have:

$$\forall T_k \in \{T \cup \tau_i\},\;\; r_j + \sum_{l=k+1}^{n+1} (E^M_{\mu(T_l)}) - r_{\mu(T_{k+1})} + (E^M_j - e_j) \leq D_j$$

which can be rewritten as

$$\forall T_k \in \{T \cup \tau_i\},\; \sum_{l=k}^{n+1}(E^M_{\mu(T_l)}) + (r_{\mu(T_k)} - r_{\mu(T_{k+1})} - e_{\mu(T_k)}) \leq D_{\mu(T_k)}$$

but $e_{\mu(T_k)} = r_{\mu(T_k)} - r_{\mu(T_{k+1})}$ and the test is reduced to:

$$\forall T_k \in \{T \cup \tau_i\},\; \sum_{l=k}^{n+1}(E^M_{\mu(T_l)}) \leq D_{\mu(T_k)} \qquad (5.21)$$

This test is pessimistic, since we are using the worst case execution time for tasks in order to calculate blocking times and schedulability. If we consider applications where execution times are *controllable*, that is, applications where we can influence in someway the time spent in the execution, we could use minimum execution times. This could be acceptable for applications where execution times are related to some quality of service, for instance performing an approximative calculus instead of an exact one or composing an image in different qualities.

On the contrary, if execution times are *uncontrollable*, then we need maximum execution times, since we must accept to work under a worst case perspective.

5.4 EDF Scheduling

Let us analyse another scheduling policy, *earliest deadline first*, EDF, which is considered to be optimal in the sense that if a set of tasks is schedulable under some policy, then it is also schedulable under EDF, [37].

Under this policy we know that tasks are chosen by the scheduler according to their deadlines, with that having the shortest deadline being in execution. The policy is generally preemptive, but we could imagine an EDF scheduler not preemptive. We will consider a one-preemption EDF scheduler.

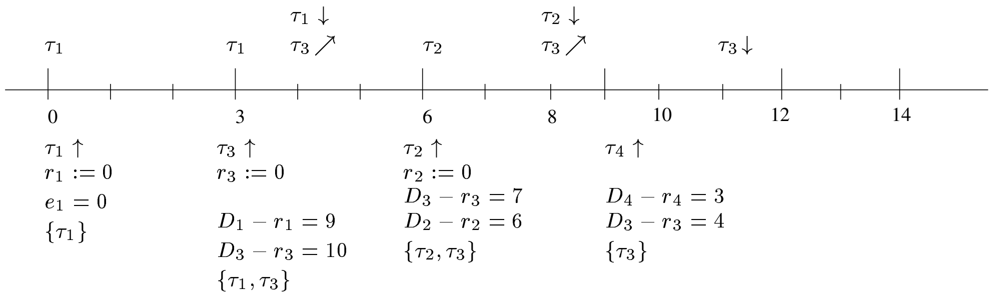


Figure 5.7: One preemption EDF Scheduler

Let T be the set of active tasks ordered by deadline; in fact, T is a queue and by convention T_n is the *head* of the queue and hence currently executing; the rest of T is the *tail.* We also assume the existence of the renaming function μ, as explained for LIFO and the universe $\mathbb{T}$ of tasks, such that $T \subseteq \mathbb{T}$.

Conceptually speaking, the EDF scheduler is quite simple, when a new task arrives it is accepted or rejected by the scheduler for schedulability reasons and if accepted it is inserted in T according to its deadline. Once a task is finished, the scheduler can choose the next one, which is that behind the head, and so on. Note that a task can be accepted and put in T in some position according to its deadline, not necessarily preempting the task in the head of T.

A one-preemption EDF scheduler works quite similarly to an EDF scheduler except that if a new task τ must preempt the currently executing one which has already been preempted, then τ is rejected even if the whole system is schedulable. Once again, the reason to do this is our manipulation of clocks.

Example 5.4 *Let us consider a set* $\mathbb{T} = \{\tau_1(4, 12), \tau_2(2, 6), \tau_3(5, 10), \tau_4(1, 3)\}$. *In figure 5.7 we see the behaviour under a one preemption* EDF *scheduling policy. Some remarks:*

- *At time* $t = 3$, $\tau_3 \uparrow$ *but its deadline (10) is longer than* τ_1*'s, so it waits in the queue.*
- *At time* $t = 4$, τ_1 *finishes and* τ_3, *gains the processor.*
- *At time* $t = 6$, $\tau_2 \uparrow$, *and its deadline (6) is shorter than* τ_3*'s, so it preempts it. This is the first preemption for* τ_3 *since it is its first execution.* τ_3 *rejoins the queue.*
- *At time* $t = 8$ τ_2 *finishes its execution and* τ_3 *resumes its execution.*
- *At time* $t = 9$ *task* τ_4 *arrives and its deadline (3) is shorter than* τ_3*'s (4) so it should preempt it* but τ_3 *had already been preempted, son our scheduler rejects* τ_4.
- *At time* $t = 11$ τ_3 *finishes.*

As usual, we distinguish the arrival of a task $\tau \uparrow$, from the resuming of a task $\tau \nearrow$. Note that in our one-preemption EDF scheduler is not optimal, since the system is feasable (we could have accepted τ_4) but we rejected it.

5.4.1 EDF Transition Model

We model an EDF application as a transition system $\mathcal{S}, T, \rightarrow)$ composed of:

1. A collection of states $\mathcal{S} = (S, \vec{e}, \vec{r}, \vec{p}, \vec{\dot{e}}, \vec{w})$, where S, $\vec{e}$, $\vec{r}$, $\vec{p}$, $\vec{\dot{e}}$ have the same meaning as for LIFO and $\vec{w}$ is an auxiliary vector of clocks, $\vec{w}_j$ notes the time when task τ_j begins its execution, which is different from its released (or arrival) time; note that in LIFO scheduling, the most recent arrived task preempts the executing one, so, immediately accepted, a task begins execution. Under EDF scheduling this is not the case, since an accepted task may go somewhere in the queue, being its execution delayed until more urgent tasks finish their executions. Clocks w's will serve to note this gap in time.
2. A collection of active tasks T
3. A transition relation $\rightarrow$.

We introduce the operations in our transition system; we note as τ_c the currently executing task, that is $\tau_c = \mathbf{exec}(T) = \tau_{\mu(T_n)}$

- Task arrival, $\tau_i\!\uparrow$ (remember: $\tau_i \in \mathbb{T}$ and $\mathbf{accept}(\mathcal{S}, T, \tau_i)$):

$$(\mathcal{S}, T,) \overset{\tau_i\uparrow}{\rightarrow} (\mathcal{S}', T')$$

where

$T' \equiv T \oplus \tau_i$ and $\oplus$ is an ordered insert operation over T for τ_i according to its deadline. $\mathcal{S}' = (S', \vec{e'}, \vec{r'}, \vec{p'}, \vec{\dot{e}'}, \vec{w'})$ is defined as:

$$\vec{e'}_j = \begin{cases} \vec{e}_j & \text{if } j \neq i \\ 0 & \text{for } j = i \wedge D_c - r_c > D_i \text{ (Execute } \tau_i) \\ \perp & \text{otherwise} \end{cases}$$

$$\vec{w'}_j = \begin{cases} \vec{w}_j & \text{if } j \neq i \\ 0 & \text{if } j = i \wedge D_c - r_c > D_i \text{ (Execute } \tau_i) \\ \perp & \text{otherwise} \end{cases}$$

$$\vec{r'}_j = \begin{cases} \vec{r}_j & \text{if } j \neq i \\ 0 & \text{otherwise} \end{cases}$$

$$\vec{p'}_j = \begin{cases} i & \text{if } j = c \wedge D_c - r_c > D_i \text{ (} \tau_i \dagger \tau_c) \\ 0 & \text{if } j = i \\ \vec{p}_j & \text{otherwise} \end{cases}$$

$$\vec{\dot{e}'}_j = \begin{cases} 1 & \text{if } j = i \wedge D_c - r_c > D_i \text{ (Execute } \tau_i) \\ 0 & \text{if } j = i \wedge D_c - r_c \leq D_i \text{ (Do not start } \tau_i) \\ 0 & \text{if } j = c \wedge D_c - r_c > D_i \text{ (Stop } \tau_c) \\ \vec{\dot{e}}_j & \text{otherwise} \end{cases}$$

- Task completion: $T_c\!\downarrow$

$$(\mathcal{S}, T) \overset{T_c\downarrow}{\rightarrow} (\mathcal{S}', T')$$

where $T' = \mathbf{tail}(T)$ and

$$\vec{e'}_j = \begin{cases} \perp & \text{if } j = c \\ \vec{e}_j & \text{otherwise} \end{cases}$$

$$\vec{w'}_j = \begin{cases} \perp & \text{if } j = c \\ \vec{w}_j & \text{otherwise} \end{cases}$$

$$\vec{e'_j} = \begin{cases} \perp & \text{if } j = c \\ 0 & \text{otherwise} \end{cases}$$

$$\vec{r}_j = \begin{cases} \perp & \text{if } j = c \wedge \nexists p_{\mu(T_k)} = c,\ 1 \leq k < n \\ \vec{r}_j & \text{otherwise} \end{cases}$$

Variable $\vec{p}$ remains unchanged.

- Task resumption: $\tau_i \nearrow$ (we assume $\tau_i = \mathbf{top}(T)$, arrived and eventually preempted in the past).

$$(\mathcal{S}, T) \xrightarrow{\tau_i \nearrow} (\mathcal{S}', T)$$

where

$$\vec{e'}_j = \begin{cases} 0 & \text{if } j = i \wedge p_i = 0 \ (\tau_i \text{ was never executed}) \\ \vec{e}_j & \text{otherwise} \end{cases}$$

$$\vec{w'}_j = \begin{cases} 0 & \text{if } j = i \wedge p_i = 0 \ (\tau_i \text{ was never executed}) \\ \vec{w}_j & \text{otherwise} \end{cases}$$

$$\vec{e'_j} = \begin{cases} 1 & \text{if } j = i \\ 0 & \text{otherwise} \end{cases}$$

Variables $\vec{p}$ and $\vec{r}$ remain unchanged.

- Time passing: δ is an elapsed time not enough to finish the current executing task.

$$(\mathcal{S}, T) \xrightarrow{\delta} (\mathcal{S}', T)$$

where

$$\vec{e'}_j = \begin{cases} \vec{e}_j + \delta & \text{if } j = c \\ \vec{e}_j & \text{otherwise} \end{cases}$$

and

$$\vec{w'}_j = \begin{cases} \vec{w}_j + \delta & \text{if } j = c \\ \vec{w}_j + \delta & \text{if } \vec{p}_j \neq 0 \\ \perp & \text{otherwise} \end{cases}$$

$\vec{r'}_j = \vec{r}_j + \delta \ \forall \tau_j, \tau_j \in T$ and variables $\vec{p}$ and $\vec{e}$ remain unchanged.

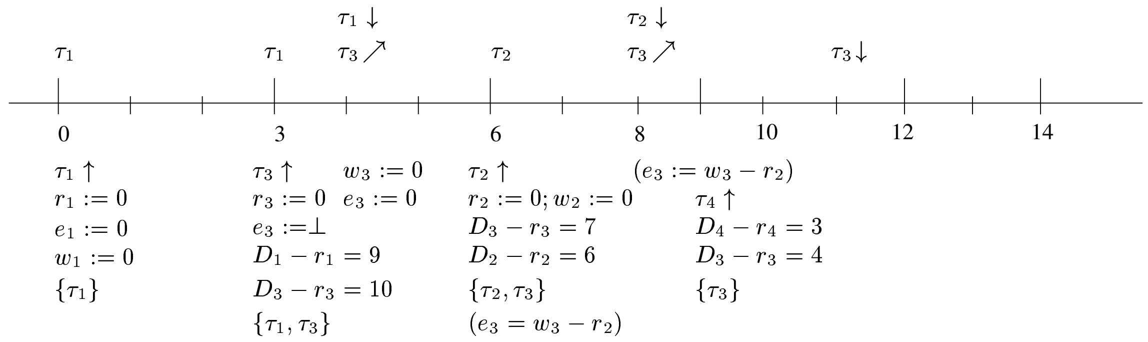

τ_1
τ_1
$\tau_1 \downarrow$ $\tau_3 \nearrow$
τ_2
$\tau_2 \downarrow$ $\tau_3 \nearrow$
$\tau_3 \downarrow$
0
3
6
8
10
12
14
$\tau_1 \uparrow$ $r_1 := 0$ $e_1 := 0$ $w_1 := 0$ $\{\tau_1\}$
$\tau_3 \uparrow$ $r_3 := 0$ $e_3 := \perp$ $D_1 - r_1 = 9$ $D_3 - r_3 = 10$ $\{\tau_1, \tau_3\}$
$w_3 := 0$ $e_3 := 0$
$\tau_2 \uparrow$ $r_2 := 0; w_2 := 0$ $D_3 - r_3 = 7$ $D_2 - r_2 = 6$ $\{\tau_2, \tau_3\}$ $(e_3 = w_3 - r_2)$
$(e_3 := w_3 - r_2)$
$\tau_4 \uparrow$ $D_4 - r_4 = 3$ $D_3 - r_3 = 4$ $\{\tau_3\}$

Figure 5.8: Usage of difference constraints

For the instant being, our operations do not show the utility of defining the new auxiliary clocks $\vec{w}$; although this is explained in the next section, let us give an example of their usage.

The automata model defined behind our transition system is a SWA where clocks $\vec{e}$ are stopped at preemption time. We want to eliminate this operation and replace it difference constraint using $\vec{w}$, as we have done for a LIFO scheduler.

In figure 5.8 we show example 5.7 using $\vec{w}$; we can see that:

- At time $t = 3$, $\tau_3 \uparrow$, $p_3 = r_3 := 0, w_3 = e_3 = \perp$, and τ_3 joins the queue.
- At time $t = 4$, $\tau_3 \nearrow$ and we set $w_3 := 0$ (note $r_3 = 1$).
- At time $t = 6$, $\tau_2 \uparrow$ and $\tau_2 \dagger \tau_3$; we see that e_3 can be expressed as the difference $w_3 - r_2$ and we see the utility of variable $\vec{w}$, since we could not express the value e_3 as $r_3 - r_2$, as we have done for LIFO, since τ_3 arrived and was not immediately executed; we need another clock to mark the first execution of τ_3. Observe that $p_3 := 2$.
- At time $t = 8$, $\tau_2 \downarrow$ and $\tau_3 \nearrow$; e_3 is recovered from the invariant difference $w_3 - r_2$.
- At time $t = 9$, $\tau_4 \uparrow$ and even if its deadline priority is shorter than τ_3's, it cannot preempt it, ($p_3 = 2 \neq 0$).
- At time $t = 11$, τ_3 finishes.

5.4.2 EDF Admittance Test

As in LIFO, each time a new task, say τ_i, arrives, we perform an acceptance test according to EDF and one-preemption policy. For EDF we propose:

$$\textbf{accept}_{\text{EDF}}(T, \tau_i) \equiv \neg\textbf{active}(T, \tau_i) \wedge \neg\textbf{preempted}(T, \tau_i)$$

that is, we do not accept a task if:

- There is an active instance of the same task:

$$\textbf{active}(T, \tau_i) \equiv (\exists\ T_k \in T.\ 1 \leq k \leq n)(\mu(T_k) = i \vee p_{\mu(T_k)} = i)$$

- It will preempt an already preempted task (in fact τ_c) :

$$\textbf{preempted}(T, \tau_i) \equiv p_c \neq 0 \wedge D_c - r_c > D_i$$

The first term, rejects a new instance of an uncompleted task or a task whose release clock is still active; the second one deals with the one preemption hypothesis under EDF which is rather tricky, since a new task may have a shorter deadline than the currently executing one, but the latter has already been preempted in the past, so the new task is rejected (even if there is enough time to execute it) or the new task may go beneath τ_c (which was not possible under LIFO).

Later, we give a refinement of this admittance test, considering deadlines, execution times and system state.

5.4.3 Properties of EDF scheduler

Let T be the set of active tasks, with $\tau_c = \tau_{\mu(T_n)} = \textbf{head}(T)$ the task under execution. We can enumerate the following properties:

1. if $p_j = 0 \Rightarrow e_j = w_j, \forall \tau_j, \tau_j \in T$
2. if $\exists\, p_j = c \Rightarrow e_j = w_j - r_{p_j} \equiv e_j = w_j - r_c$
3. if $\tau_c \nearrow \wedge p_c = 0 \Rightarrow e_c = w_c = \perp \wedge \neg \exists p_j = c$
4. $\forall p_j \neq 0, j \neq c \;\Rightarrow e_j = w_j - r_{p_j}$

Property 1 says that a task τ_j in T which has never been preempted respects $e_j = w_j = \perp$. In fact, if $\tau_j \in T$ and $\tau_j \neq \tau_c$, then τ_j arrived in the past, its deadline was not urgent enough to preempt the currently executing task, and it was put in the queue according to its deadline with $e_j = w_j = \perp$; on the contrary if $\tau_j = \tau_c$ and it has never been preempted, then $e_j = w_j \geq 0$.

Property 2 is a consequence of preemption; if $p_j = c$ it means that τ_c preempted τ_j, in fact, when τ_j was running, $p_j = 0$ (one-preemption assumption) which implies $e_j = w_j$ (property 1); as τ_j was preempted by τ_c, its computation time can be put as $e_j = w_j - r_c$ (since $r_c = 0$ when τ_c arrived). As time passes, while $\dot{e}_i = 0$, $e_j = (w_j + \delta) - (r_c + \delta) = w_j - r_c$. This property shows that execution times can be re-written as differences of some clocks for those stopped tasks.

Property 3 is a consequence of the EDF policy. It means that τ_c resumes but it had never been preempted; so the scenario is as follows: when τ_c arrived, its deadline was longer than that of the currently executing task and hence, it was put in the queue, but never executed, so $e_c = w_c = \perp$; as it has not executed, it could not have preempted any other task (in particular the one executing at its arrival time). Note that τ_c can be preempted during its execution.

The last property 4 is our execution invariant for EDF, which says that for all preempted tasks, (except the current executing task), we can express its computed time as a difference. This is an extension of property 2 and 3, since there may be tasks in T never preempted and never executed. This is a great difference compared to LIFO. This property can be put as:

$$I_{\mathrm{EDF}}(T) \equiv \bigwedge_{\tau_j \in T, p_j \neq 0} e_j = w_j - r_{p_j}$$

recall that $e_j = w_j = \perp$, if $\tau_j \in T \wedge p_j = 0$.

For the current executing task, even if preempted in the past, property 4 does not hold since its execution clock is running.

Remark Note that property 4 obliges to keep clock r_{p_j} even if task τ_{p_j} has already finished; for the same reason, we cannot accept a new instance of this task if τ_j is still active. This is a restriction of our model (taken into account by the admittance test), which could be relaxed if we create a "preemptable clock" for each task instance that preempts; a rather costly solution.

We conclude the section with a theorem, analogous to that give for a LIFO scheduler, without proof, since we will give a detailed proof of the general case in section 5.5.

Theorem 5.2 *The symbolic reachability graph of a system of tasks for an* EDF *scheduler satisfying the one-preemption constraint is finite.*

5.4.4 Refinement of EDF Admittance Test

As in LIFO, each time a new task, say τ_i, arrives, we perform an acceptance test according to EDF and one-preemption policy. For EDF we propose:

$$\mathbf{accept}_{\mathrm{EDF}}(T,\tau_i) \equiv \neg\mathbf{active}(T,\tau_i) \wedge neg\mathbf{preempted}(T,\tau_i) \wedge \mathbf{schedulable}(T,\tau_i)$$

The first two predicates have already been explained; in this section, we develop a test regarding execution times, deadlines and system state. The question is 'will the new arrived task, if accepted, cause other tasks in T miss their deadlines?

In principle, the predicate **schedulable** is:

$$\mathbf{schedulable}(T,\tau_i) \equiv \forall\tau_j \in \{T \cup \tau_i\},\ r_j + B_j + (E_j^M - e_j) \leq D_j \tag{5.22}$$

where the blocking time for a task τ_j which is in position k of T is expressed as:

$$B_j^{EDF} = \sum_{l>k}^{n+1}(E^M_{\mu(T_l)} - e_{\mu(T_l)})$$

Under the EDF scheduler we know that for those $\tau_j \in T$ preempted in the past, we have $e_j = w_j - r_{p_j}$ and for those τ_j's never preempted, $e_j = w_j = \perp$, so the precedent expression can be split into:

$$\begin{aligned} B_j^{EDF} \quad = \quad & \sum_{l=k+1, p_{\mu(T_l)}\neq 0}^{n-1} (E^M_{\mu(T_l)} - e_{\mu(T_l)}) + (E^M_{\mu(T_n)} - e_{\mu(T_n)}) + \\ & \sum_{l=k+1, p_{\mu(T_l)}=0}^{n-1} E^M_{\mu(T_l)} + E^M_{\mu(T_{n+1})} \end{aligned}$$

using the equality for preempted task we have:

$$\begin{aligned} B_j^{EDF} \quad = \quad & \sum_{l=k+1, p_{\mu(T_l)}\neq 0}^{n-1} (E^M_{\mu(T_l)} - (w_{\mu(T_l)} - r_{p_{\mu(T_l)}})) + (E_{\mu(T_n)} - e_{\mu(T_n)}) + \\ & \sum_{l=k+1, p_{\mu(T_l)}=0}^{n-1} E^M_{\mu(T_l)} + E^M_{\mu(T_{n+1})} \end{aligned}$$

$$B_j^{EDF} = \sum_{l=k+1}^{n+1} E^M_{\mu(T_l)} - \sum_{l=k+1, p_{\mu(T_l)} \neq 0}^{n-1} (w_{\mu(T_l)} - r_{p_{\mu(T_l)}}) - e_{\mu(T_n)} \tag{5.23}$$

Unfortunately we can say nothing about the second term in 5.23, so we will try to find some bounds for this term in order to get necessary or sufficient conditions for our schedulability test. We deal with two facts:

1. In 5.23 we have
$$B_j^{EDF} \leq \sum_{l=k+1}^{n+1} E^M_{\mu(T_l)} \tag{5.24}$$
since all terms representing execution times are positive. This fact gives a sufficient condition for the admission test; which is too conservative but safe, in the sense that if we accept τ_i we know all tasks in T, including the new one, will be scheduled within their deadlines.

2. Using minimum execution times:
$$B_j^{EDF} \geq \sum_{l=k+1}^{n+1} E^m_{\mu(T_l)} \tag{5.25}$$
since minimum execution times represent the fastest execution, this bound is a necessary condition, more laxative but unsafe. If after considering minimum execution times, the test of schedulability is **not** satisfied, then no admission is possible; if the test **is** satisfied, then we can accept but we know that there may be some executions leading to error states and hence we need some dynamic control.

Reconsidering our schedulability test 5.22:

$$\forall\, \tau_j \in \{T \cup \tau_i\},\ r_j + \sum_{l=k+1}^{n+1} E^M_{\mu(T_l)} - \sum_{l=k+1, p_{\mu(T_l)} \neq 0}^{n-1} (w_{\mu(T_l)} - r_{p_{\mu(T_l)}}) + (E^M_j - e_j) \leq D_j$$

$$\forall\, \tau_j \in \{T \cup \tau_i\},\ \sum_{l=k}^{n+1} E^M_{\mu(T_l)} - \sum_{l=k+1, p_{\mu(T_l)} \neq 0}^{n-1} (w_{\mu(T_l)} - r_{p_{\mu(T_l)}}) + (r_j - e_j) \leq D_j \tag{5.26}$$

Now we analyse 5.26 to find some bounds; we consider two cases:

1. $p_j = 0$
   - for $k < n$, we know $e_j = 0$ and so 5.26 becomes:
   $$\forall\, \tau_j \in \{T \cup \tau_i\},\ \sum_{l=k}^{n+1} E^M_{\mu(T_l)} - \sum_{l=k+1, p_{\mu(T_l)} \neq 0}^{n-1} (w_{\mu(T_l)} - r_{p_{\mu(T_l)}}) + r_j \leq D_j \tag{5.27}$$
   - for $k = n$, we know $e_j = w_j \geq 0$ and expression 5.26 is:
   $$(E^M_j + E^M_{\mu(T_{n+1})} - (w_j - r_j) \leq D_j \tag{5.28}$$

2. $p_j \neq 0$, we know $e_j = w_j - r_{\mu(T_{k+1})}$ so 5.26 becomes:

$$\forall\, \tau_j \in \{T \cup \tau_i\},\ \sum_{l=k}^{n+1} E^M_{\mu(T_l)} - \sum_{l=k, p_{\mu(T_l)} \neq 0}^{n-1} (w_{\mu(T_l)} - r_{p_{\mu(T_l)}}) + r_j \leq D_j \tag{5.29}$$

Now considering our bounds 5.24 and 5.25 we have:

- $$\forall\, \tau_j \in \{T \cup \tau_i\},\ \sum_{l=k}^{n+1} E^M_{\mu(T_l)} - \sum_{l=k, p_{\mu(T_l)} \neq 0}^{n-1} (w_{\mu(T_l)} - r_{p_{\mu(T_l)}}) + r_j \leq \underbrace{\sum_{l=k}^{n+1} E^M_{\mu(T_l)} + r_j}_{\alpha} \tag{5.30}$$

  If $\forall\, \tau_j \in \{T \cup \tau_i\}, \alpha \leq D_j$ then we can accept the new task τ_i. On the contrary, we reject it, but we know we are being too restrictive.

- $$\forall\, \tau_j \in \{T \cup \tau_i\}, \sum_{l=k}^{n+1} E^M_{\mu(T_l)} - \sum_{l=k, p_{\mu(T_l)} \neq 0}^{n-1} (w_{\mu(T_l)} - r_{p_{\mu(T_l)}}) + r_j \geq \underbrace{\sum_{l=k}^{n+1} E^m_{\mu(T_l)} + r_j}_{\beta} \tag{5.31}$$

  Once again, if $\exists\, \tau_j \in \{T \cup \tau_i\}, \beta > D_j$, we do not accept τ_i.

These hypothesis could be used according to the nature of *execution times*; if execution times are *controllable*, that is we can influence the time spent in the execution, we could use minimum execution times. This could be acceptable for applications where execution times are bound to some quality of service, for instance performing an approximative calculus instead of an exact one or composing an image in different qualities.

On the contrary, if execution times are *uncontrollable*, then we need maximum execution times, since we must accept to work under a worst case perspective.

5.5 General schedulers

In this section we consider general scheduling policies, that is, preemptive schedulers based on some priority assignment mechanism which can be fixed or dynamic. We will relax the constraint of one-preemption imposed to LIFO and EDF schedulers and we consider uncertain, but bounded, execution times.

Instead of using a stopwatch automaton as we have done in the previous sections, we use a model based on timed automata as shown in figure, 5.9 where to each task τ_i we add a clock w_i which initially counts the accumulated computed time for a task. The main idea is to replace a stopped clock by an operation of difference of two running clocks, to keep track of already executed time.

Clocks w's are used as follows: preemption is only possible at arrival of a new task, say τ_j and each time a task $\tau_j \dagger \tau_i$, e_i is accumulated in w_i, τ_j gains processor and when τ_i is resumed, we recover e_i as the difference $w_i - r_j$; clock e_i is then never stopped but updated. This procedure relaxes the one preemption hypothesis but still obliges to keep clock r_j even if task τ_j has finished its execution and hence it is not active.

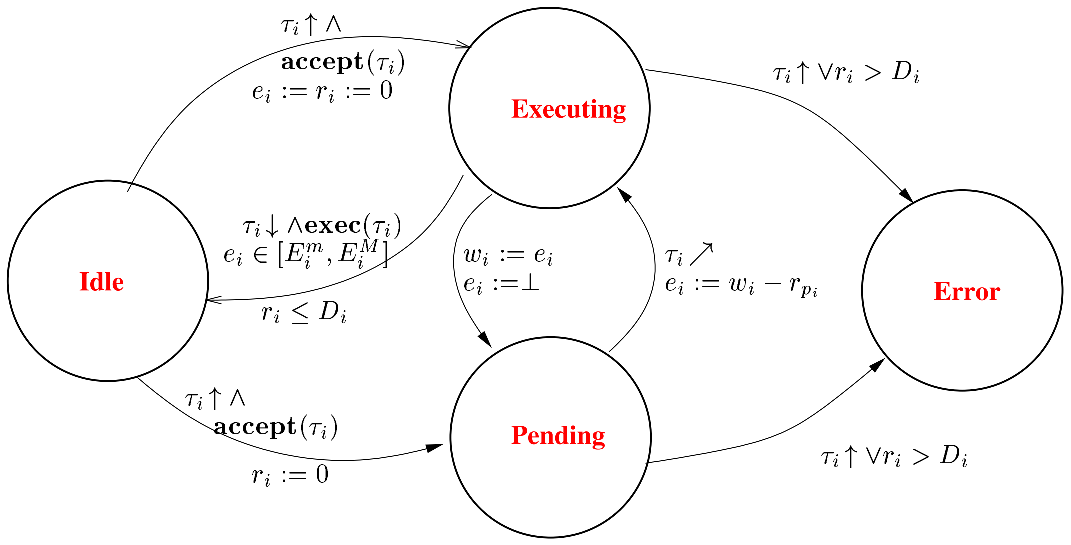

Idle
Executing
Pending
Error
$\tau_i\!\uparrow \wedge$ **accept**(τ_i) $e_i := r_i := 0$
$\tau_i\!\downarrow \wedge$**exec**$(\tau_i)$ $e_i \in [E_i^m, E_i^M]$ $r_i \leq D_i$
$\tau_i\!\uparrow \wedge$ **accept**(τ_i) $r_i := 0$
$w_i := e_i$ $e_i := \perp$
$\tau_i \nearrow$ $e_i := w_i - r_{p_i}$
$\tau_i\!\uparrow \vee r_i > D_i$
$\tau_i\!\uparrow \vee r_i > D_i$

Figure 5.9: Automaton for a General Scheduler

Example 5.5 *Let us consider $\mathbb{T} = \{\tau_1(4,13), \tau_2(5,10), \tau_3(2,10), \tau_4(3,6)\}$; we show how the introduction of w's clocks can help to relax the constraint of one-preemption under an* EDF *scheduler, see figure 5.10.*

- At time $t = 0$, $\tau_1\!\uparrow$, $r_1 = e_1 := 0$
- At time $t = 2$, $\tau_2\!\uparrow$ and its deadline 10 is shorter than τ_1's (11), so, τ_1 is preempted and joins the queue; $w_1 := e_1 = 2$. From there on the value of $e_1 = w_1 - r_2$.
- At time $t = 4$, $\tau_3\!\uparrow$ and its deadline 10 is longer than τ_2's, so it joins the queue (after τ_1).
- At time $t = 5$, $\tau_4\!\uparrow$, its deadline 6 is shorter than τ_2's, which is preempted and we set $w_2 := e_2 = 3$; from there on $e_2 = w_2 - r_4$.
- At time $t = 8$, $\tau_4\!\downarrow$ and e_2 is updated as $w_2 - r_4 = 6 - 3$; taui2 resumes execution.
- The rest of the tasks proceed in a similar manner. Note that at time $t = 12$, τ_3 gains the processor for the first time, e_3 and w_3 are indefined.

5.5.1 Transition Model

Formally the transition system is of the form $(\mathcal{S}, Q, \rightarrow)$ composed of discrete events and time passing transitions, as already mention in the precedent sections.

- $\mathcal{S} = (S, \vec{e}, \vec{r}, \vec{p}, \vec{w})$, where S, $\vec{e}$, $\vec{r}$ and $\vec{p}$ have the same meaning as in EDF and $\vec{w}$ is the auxiliary vector to reconstruct the execution times after preemption,
- Q is a queue of tasks, with the usual operations: $\oplus$, for adding an element, **pop** to remove the element at the head, **top**, to choose the task at the head.

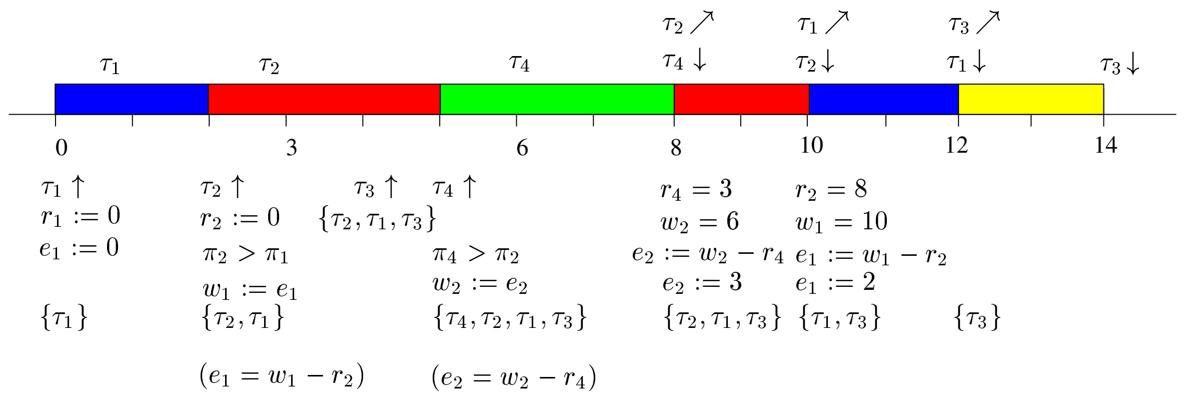


Figure 5.10: General EDF Scheduler

- $\to$ is the transition relation.

We list the operations over our transition system; recall that the arrival of a task is captured by the scheduler, who decides over the admission. We assume also that the schedulers 'knows' the priority of each task (dynamic or fixed); of course, the currently executing task, denoted τ_c, is on the head of the queue and has the highest priority; priority of task τ_i is noted π_i, as usual; the operation $\oplus$ works on a queue according to a scheduling policy.

- Task arrival, $\tau_i\uparrow$ $(\tau_i \in \mathbb{T})$:

$$(\mathcal{S}, Q) \overset{\tau_i\uparrow}{\to} (S', Q')$$

where

$Q' = Q \oplus \tau_i$.

$$\vec{e'}_j = \begin{cases} \vec{e}_j & \text{if } j \neq i \\ 0 & \text{if } j = i \wedge \pi_i > \pi_c \ (\mathbf{exec}(\tau_i)) \\ \perp & \text{otherwise} \end{cases}$$

$$\vec{r'}_j = \begin{cases} 0 & \text{if } j = i \\ \vec{r}_j & \text{otherwise} \end{cases}$$

$$\vec{p'}_j = \begin{cases} i & \text{if } j = c \wedge \pi_i > \pi_c \ (\tau_i \dagger \tau_c) \\ 0 & \text{if } j = i \text{ (no task preempted } \tau_i) \\ \vec{p}_j & \text{otherwise} \end{cases}$$

$$\vec{w'}_j = \begin{cases} \vec{e}_j & \text{if } j = c \wedge \pi_i > \pi_c \ (\tau_i \dagger \tau_c) \\ \perp & \text{if } j = i \\ \vec{w}_j & \text{otherwise} \end{cases}$$

Note that $w_c := e_c$ if τ_c is preempted by τ_i and $w_i = \perp$.

- Task completion: $\tau_c\!\downarrow$

$$(\mathcal{S}, Q) \xrightarrow{\tau_c\downarrow} (S', Q')$$

where $Q' = \mathbf{pop}(Q)$ and

$$\vec{e'}_j = \begin{cases} \bot & \text{if } j = c \\ \vec{e}_j & \text{otherwise} \end{cases}$$

$$\vec{w'}_j = \begin{cases} \bot & \text{if } j = c \\ \vec{w}_j & \text{otherwise} \end{cases}$$

$$\vec{r}_j = \begin{cases} \bot & \text{if } j = c \wedge \nexists p_{\mu(T_k)} = c,\ 1 \leq k < n \\ \vec{r}_j & \text{otherwise} \end{cases}$$

Variable $\vec{p}$ remains unchanged.

- Task resumption: $\tau_i\!\nearrow$ (we assume $\tau_i = \mathbf{top}(Q)$ is a task which regains the processor).

$$(S, Q) \xrightarrow{\tau_i\nearrow} (S', Q)$$

where

$$\vec{e'}_j = \begin{cases} \vec{w}_j - \vec{r}_{\vec{p}_j} & \text{if } j = i \wedge \vec{p}_j \neq 0 \ (\tau_{\vec{p}_j} \dagger \tau_i) \\ 0 & \text{if } j = i \wedge \vec{p}_j = 0 \ (\tau_i \text{ was never preempted}) \\ \vec{e}_j & \text{otherwise} \end{cases}$$

Variables $\vec{r}$, $\vec{p}$ and $\vec{w}$ remain unchanged.

- Time passing: δ is an elapsed time not enough to finish the current executing task, τ_c.

$$(S, Q, \vec{e}, \vec{r}, \vec{p}, \vec{w}) \xrightarrow{\delta} (S', Q, \vec{e'}, \vec{r'}, \vec{p}, \vec{w'})$$

where

$$\vec{e'}_j = \begin{cases} \vec{e}_j + \delta & \text{if } j = c \\ \vec{e}_j & \text{otherwise} \end{cases}$$

and $\vec{r'}_j = \vec{r}_j + \delta$ and $\vec{w'}_j = \vec{w}_j + \delta\ \forall \tau_j \in Q$

Remark I Note that clock w_i is initially set to bottom at τ_i arrival, and it is updated to e_i if this task is preempted, so saving the cumulated executing time; from there on w_i grows (while e_i is $\bot$) and when τ_i regains processor its cumulated time is recovered from the difference between w_i and the released clock of the preempter (kept in $\vec{p}_i$). This implies that released clocks cannot disappear until the preempted task regains the processor. This condition must be tested at admission time of a new task. Figure 5.11 shows the evolution of clocks.

A possibly more elegant way of solving the problem consists in systematically adding the new variable h_i for each task, and use it in the time-invariant equations of the form $e_c = w_c - h_i$. In this case, the r variables are eliminated at completion time but many 'instances of h' may be necessary

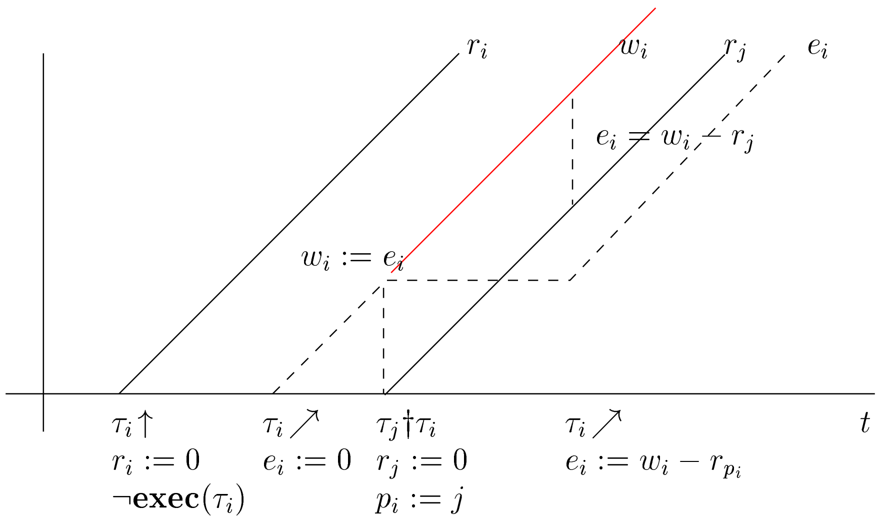


Figure 5.11: Evolution of $\vec{w}$ and $\vec{e}$

to be created as τ_i may be a very eager task with high priority preempting different tasks at each arrival. This approach unnecessary complicates the proofs (as it requires carrying through additional invariants). Besides, it is not very useful in practice as it augments the complexity by increasing the number of clocks.

Remark II We will show that the fact of simulating a stopped clock e_i by a difference constraint of the form $w_i - r_{p_i}$, both running does not disturbe the semantics of the systems; indeed we will prove that the relationships where e_i is involved can be replaced by this expression while e_i is stopped and still the problem of schedulability, view as the problem of reachability of an error state is decidable.

5.5.2 Properties of a General Scheduler

Let Q be the queue of active tasks, $Q \in \mathbb{T}$ where $\tau_c = \mathbf{top}(Q)$. We can enumerate the following properties:

1. if $p_j = 0 \Rightarrow e_j = w_j, \forall \tau_j, \tau_j \in Q$
2. if $\exists\, p_j = c \Rightarrow e_j = w_j - r_c$
3. if $\tau_c \nearrow \wedge p_c = 0 \Rightarrow e_c = w_c = \perp \wedge \neg \exists p_j = c,$ for any $\tau_j \in Q$
4. $\forall \tau_j \in \{Q - \mathbf{top}(Q)\} \wedge p_j \neq 0 \Rightarrow e_j = w_j - r_{p_j}$

Properties 1, 2 and 3 are completely analogous to the corresponding EDF scheduler properties. The last property 4 can be reformulated to create our **general execution invariant**:

$$I_{\mathrm{SCH}}(Q) \equiv \bigwedge_{\tau_j \in Q', p_j \neq 0} e_j = w_j - r_{p_j} \wedge \bigwedge_{\tau_j \in Q', p_j = 0} e_j = \perp$$

where $Q' = \mathbf{pop}(Q)$.

The invariant says that for those tasks waiting for execution and preempted their cumulated executed time can be express as a difference of clocks; evidently, for those tasks never executed at all their cumulated executed time is unknown.

5.5.3 Schedulability Analysis

Let Ψ as explained in LIFO analysis and let ϕ be a constraint characterizing a set of states. We define $\tau_i\!\uparrow(\phi)$ to be the set of states reached when task τ_i arrives, that is:

$$\tau_i\!\uparrow(\phi) = \{s' : \exists s \in \phi.\ s \overset{\tau_i\uparrow}{\rightarrow} s'\}$$

Let ϕ be of the form $I_{\text{SCH}}(Q) \wedge \psi$, with $\psi \in \Psi$; that is ϕ characterizes a state with the execution invariant for all waiting tasks and clock relationships expressed as differences.

Without loss of generality, we can assume that either:

1. τ_i is rejected: $\psi \implies \neg\mathbf{accept}(\psi, Q, \tau_i)$, in which case we have $\tau_i\!\uparrow\!(\phi) \equiv \phi$.
2. τ_i is accepted: $\psi \implies \mathbf{accept}(\psi, Q, \tau_i)$. Does $\tau_i \dagger \tau_c$?

$$\text{if } \neg\tau_i\dagger\tau_c, \text{ then } \tau_i\!\uparrow(\phi) \equiv I_{\text{SCH}}(Q \oplus \tau_i) \wedge r_i = 0 \wedge e_i = \perp \wedge w_i = \perp \wedge \psi \equiv I_{\text{SCH}}(Q') \wedge \psi$$

$$\text{if } \tau_i\dagger\tau_c, \text{ then } \tau_i\!\uparrow(\phi) \equiv I_{\text{SCH}}(Q \oplus \tau_i) \wedge r_i = 0 \wedge e_i = 0 \wedge \psi[e_c := w_c] \equiv I_{\text{SCH}}(Q') \wedge \psi'$$

where $\psi[e_c := w_c]$ is the substitution of e_c for w_c in ψ. In summary:

$$\tau_i\!\uparrow(\phi) \equiv I_{\text{SCH}}(Q') \wedge \psi'$$

Hence, we have:

Proposition 5.5 *$\tau_i\!\uparrow\!(\phi)$ has the same structure than ϕ, that is, it is the conjunction of an execution invariant and a formula in Ψ.*

Now, let $\phi\nearrow$ be the set of states reached from ϕ by letting time advance, that is:

$$\phi\nearrow = \{s' : \exists s \in \phi, \delta \geq 0. s \xrightarrow{\delta} s'\}$$

Clearly, if ϕ is of the form $I_{\text{SCH}}(Q) \wedge \psi$, we have that

$$\phi\nearrow = (I_{\text{SCH}}(Q) \wedge \psi)\nearrow \equiv I_{\text{SCH}}(Q) \wedge \psi\nearrow$$

As $\psi \in \Psi$ over clocks in $\mathcal{S}$, we can express these differences in a DBM. The following proposition gives this equivalence:

Proposition 5.6 *Let ϕ be of the form $I \wedge M$, where M is a* DBM *and I is an execution invariant under* SCH. *Then, $\tau_i\!\uparrow(\phi)$ and $\phi\nearrow$ have the same structure as ϕ.*

Thus, given a sequence σ of task arrivals the set of reached states $\sigma(\phi)$ can be represented by the conjunction of the execution invariant $I_{\mathrm{SCH}}(Q)$, characterizing the already executed time of the suspended tasks and a DBM M, characterizing the relationships between the corresponding r's and w's clocks.

A DBM M has the following form (since $e_n = r_n$ we omit it; in order to facilitate comprehension, we "name" clocks according to the position of their corresponding tasks in Q):

$$\begin{array}{llllllcll}
M & u & r_1 & w_1 & r_2 & w_2 & \dots & r_n & w_n \\
u & - & M_{ur_1} & M_{uw_1} & M_{ur_2} & M_{uw_2} & \dots & M_{ur_n} & M_{uw_n} \\
r_1 & M_{r_1u} & - & - & M_{r_1r_2} & M_{r_1w_2} & \dots & M_{r_1r_n} & M_{r_1w_n} \\
w_1 & M_{w_1u} & - & - & M_{w_1r_2} & M_{w_1w_2} & \dots & M_{w_1r_n} & M_{w_1w_n} \\
r_2 & M_{r_2u} & M_{r_2r_1} & M_{r_2w_1} & - & - & \dots & M_{r_2r_n} & M_{r_2w_n} \\
w_2 & M_{w_2u} & M_{w_2r_1} & M_{w_2w_1} & - & - & \dots & M_{w_2r_n} & M_{w_2w_n} \\
\vdots & & & & & & & & \\
r_n & M_{r_nu} & M_{r_nr_1} & M_{r_nw_1} & M_{r_nr_2} & M_{r_nw_2} & \dots & - & - \\
w_n & M_{w_nu} & M_{w_nr_1} & M_{w_nw_1} & M_{w_nr_2} & M_{w_nw_2} & \dots & - & -
\end{array}$$

Once again, M is constructed in a very special way and has a particular structure. Let us analyse it:

- The new matrix M is constructed as new tasks τ_i's arrive; we denote $M' = M^{\tau_i\uparrow}$ the values in M immediately after acceptance of τ_i.
- When $\tau_i\uparrow$, we have two possible situations (assuming it is accepted):
  - $\tau_i \dagger \tau_c$, then $r_i = e_i := 0$, we need to stop e_c and create w_c with value e_c, we have that $e_c = w_c - r_i \leq M'_{w_cr_i} = M_{e_cr_i}$, but $r_i = 0$ and so $M'_{w_cr_i} = M_{e_cr_i} = M_{e_cu} = M'_{w_cu}$.
  - $\neg(\tau_i \dagger \tau_c)$, then $r_i := 0, e_i := \perp$, we have $r_c - r_i \leq M'_{r_cr_i}$, $r_i = 0$ and so we have $M'_{r_ir_c} = M_{r_cu}$. This relation is also respected in the precedent case.
- When $\tau_c\downarrow$, we have again two situations:
  - $\nexists p_j = c,\ \tau_j \in Q$:

    $$(I_{\mathrm{SCH}}(Q) \wedge \psi)[w_c := \perp, r_c := \perp] \equiv I_{\mathrm{SCH}}(\mathbf{pop}(Q)) \wedge \psi'$$
  - $\exists p_j = c,\ \tau_j \in Q$:

    $$(I_{\mathrm{SCH}}(Q) \wedge \psi)[w_c := \perp]$$
- When $\tau_i \nearrow$, once again two situations are possible:
  - $p_i = j,$:

    $$(I_{\mathrm{SCH}}(Q) \wedge \psi)[r_j := w_i - e_i] \equiv I_{\mathrm{SCH}}(\mathbf{pop}(Q)) \wedge \psi'$$
  - $p_i = 0$:

    $$I_{\mathrm{SCH}}(Q) \wedge \psi \wedge e_i := 0$$

So, the characterization of each state as time passes or new tasks arrive or resume is preserved as differences of running clocks. At each operation, the representation under a DBM keeps the structure of bounded differences

Now, *what is the structure of M after a task completion?* Is it still a DBM?

We will prove that this operation still enables us to characterize the states as bounded differences in a DBM, so establishing that $\tau_i\downarrow(\phi)$ is still the conjunction of an invariant and a formula in Ψ.

Let us expose the scenario when a task τ_i finishes. At that moment, the scheduler will choose another task, say τ to regain the processor; this task had been eventually preempted in the past by another task, say $\overline{\tau}$ and the relation $e := w - \overline{r}$ shows the computed time for τ. Clock $\overline{r}$ can now be eliminated from M and replaced by $w - e$. What happens to all differences in M where $\bar{r}$ is named?

We have the following relations involving $\bar{r}$:

1. Base relations:

$$\begin{aligned} w - \overline{r} \leq M_{w\overline{r}} &\quad\Rightarrow\quad e - u \leq M_{w\overline{r}} &(5.32)\\ \overline{r} - w \leq M_{\overline{r}w} &\quad\Rightarrow\quad u - e \leq M_{\overline{r}w} &(5.33)\\ u - \overline{r} \leq M_{u\overline{r}} &\quad\Rightarrow\quad e - w \leq M_{u\overline{r}} &\\ \overline{r} - u \leq M_{\overline{r}u} &\quad\Rightarrow\quad w - e \leq M_{\overline{r}u} & \end{aligned}$$

2. Let x be another clock different from w and u:

$$\begin{aligned} x - \overline{r} \leq M_{x\overline{r}} &\quad\Rightarrow\quad x - (w - e) \leq M_{x\overline{r}} &(5.34)\\ \overline{r} - x \leq M_{\overline{r}x} &\quad\Rightarrow\quad (w - e) - x \leq M_{\overline{r}x} &(5.35) \end{aligned}$$

We can consider that these differences can be decomposed in the following ways:

1. In 5.34 and considering 5.32:

$$\left.\begin{array}{lcl} x - w & \leq & M_{xw}\\ e - u & \leq & M_{w\overline{r}} \end{array}\right\} \Rightarrow x - (w - e) \leq M_{xw} + M_{w\overline{r}}$$

$$M_{xw} + M_{w\overline{r}} \quad \theta \quad M_{x\overline{r}} \tag{5.36}$$

2. In 5.35 and considering 5.33

$$\left.\begin{array}{lcl} w - x & \leq & M_{wx}\\ u - e & \leq & M_{\overline{r}w} \end{array}\right\} \Rightarrow (w - e) - x \leq M_{\overline{r}w} + M_{wx}$$

$$M_{\overline{r}w} + M_{wx} \quad \theta \quad M_{\overline{r}x} \tag{5.37}$$

Both expressions are not difference constraints but we will show that in 5.36 and 5.37 θ represents equality; that is, we will prove that:

$$\begin{aligned} M_{xw} + M_{w\overline{r}} &= M_{x\overline{r}} &(5.38)\\ M_{\overline{r}w} + M_{wx} &= M_{\overline{r}x} &(5.39) \end{aligned}$$

and hence they are deductible from M, no need to keep them in the $M'^{\tau_i\downarrow}$.

To prove this we will consider a task τ which regains the processor after being interrupted by another task $\bar{\tau}$, point ① in figure 5.12; a third task $\hat{\tau}$ will be used to express the evolution of differences

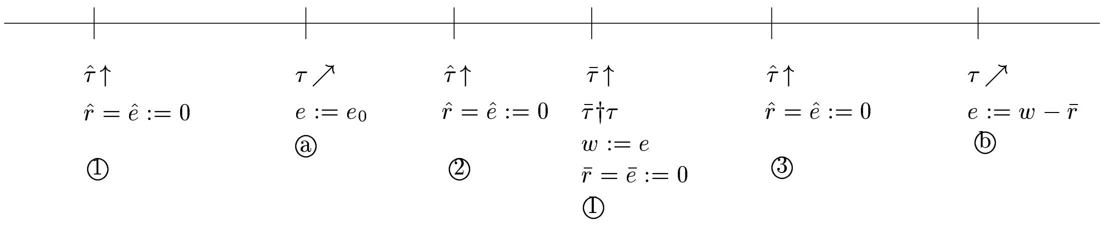


Figure 5.12: Analysis of DBM M

as τ is executing or waiting. In the figure we show the three different possibilities of arrival for such a task $\hat{\tau}$ in the system, namely ①,②,③; $\tau \nearrow$ at point ⓐ indicates the last execution for τ when it was preempted by $\bar{\tau}$; its cumulated executed time is e_0. We are analysing clock relationships when τ prepares to resume its execution (point ⓑ in figure 5.12), after it was preempted by $\bar{\tau}$

It is of extreme importance to remark two properties concerning our scenario:

- **Monotony**: clocks grow at the same rate; in our model the derivative of a clock is always 1 (if it is running) or 0 (if it is stopped).
- **Continuity**: From point ⓐ to point ⓑ clocks for τ were not reset neither updated. They were not reset, because any new instance of τ should have been rejected by the scheduler, since a previous instance is still active (and reset is only applied at task arrival). On the other hand, clocks were not updated, because the only possibility is to update e by the operation $e := w - \bar{r}$ or w by the operation $w := e$, but we are supposing that in between no resuming of τ occurs; in fact point ⓑ is the first execution after the last preemption, point ①, so no such update operation is possible.

In interval [ⓐ, ⓑ], clocks r and w are running monotonously and continuously while clock e is stopped. This means that differences such as $r - x$ and $w - x$ where x is also running, do not invalidate the respective bounds M_{rx} and M_{wx}; also, x cannot be a stopped clock, since if it were, it would be an execution clock e' of a preempted task τ' and in that case, we should have replaced it by its appropriate difference involving two continous clocks. In point ⓑ clock $\bar{r}$ can be eliminated and replaced by $w - e$ in M which leaves us with three term differences such as 5.34: we will prove that these differences can be deduced by simple bounded differences.

We know that if $\bar{\tau}\dagger\tau$ then it must be $\pi_{\bar{\tau}} > \pi_{\tau}$. At preemption time, that is when $\bar{\tau}\uparrow$, we set $w := e$, $\bar{r} = \bar{e} := 0$ and $\bar{w} := \perp$.

We note that arrival times are indeed intervals, since our execution times are unknown but bounded; remember that values in M are characterized by a super-index indicating its value at a certain moment; for instance $M_{eu}^{\bar{\tau}\uparrow}$ means "the maximum value for e at arrival of $\bar{\tau}$".

We will prove equality for expression 5.38 but it is absolutely simetric for 5.39.

Case ①

We distinguish two cases according to priority relationships; either $\pi_{\hat{\tau}} > \pi_{\tau}$ or $\pi_{\hat{\tau}} < \pi_{\tau}$

1. $\pi_{\hat{\tau}} > \pi_{\tau}$, this means that at $\tau\nearrow$, task $\hat{\tau}$ did finish its execution but there may be anohter task $\tilde{\tau}$ preempted by $\hat{\tau}$ still active, with priority $\pi_{\tilde{\tau}} < \pi_{\tau}$; under this scenario clock $\hat{r}$ is still running, but clock $\hat{w}$ has disappeared.

Figure 5.13 shows the situation graphically.

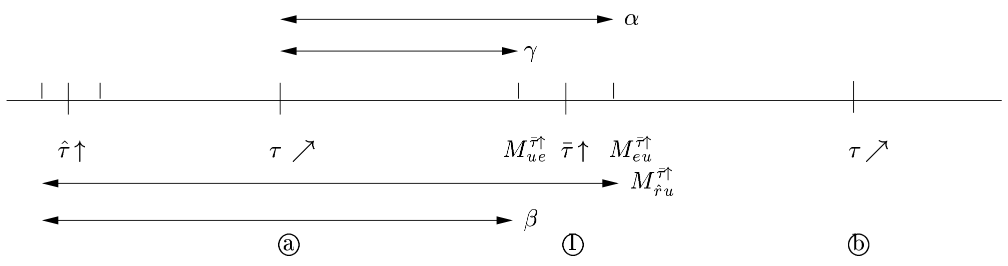


Figure 5.13: Case ① $\pi_{\hat{\tau}} > \pi_{\tau}$

$$M_{\hat{r}w}^{\tau\nearrow} + M_{w\bar{r}}^{\tau\nearrow} \theta M_{\hat{r}\bar{r}}^{\tau\nearrow}$$

- $M_{\hat{r}w}^{\tau\nearrow} = M_{\hat{r}e}^{\bar{\tau}\uparrow} = M_{\hat{r}e}^{\tau\nearrow}$
  This sucession of equalities is based on our properties of monotony and continuity; in fact, the difference $\hat{r} - w$ at $\tau\nearrow$ (point ⓑ) is the same since w was created, that is in point ① when $\bar{\tau}\uparrow$, which equals the value e; this difference is constant as both e and $\hat{r}$ were running (point ⓐ). The same reasoning as a chain of equalities is kept all over the proof.
- $M_{w\bar{r}}^{\tau\nearrow} = M_{eu}^{\bar{\tau}\uparrow} = \alpha + e_0$
- $M_{\hat{r}\bar{r}}^{\tau\nearrow} = M_{\hat{r}u}^{\bar{\tau}\uparrow}$
- In figure 5.13 we have:

$$M_{ue}^{\bar{\tau}\uparrow} = \gamma + e_0$$

and

$$M_{\hat{r}u}^{\bar{\tau}\uparrow} - \beta = \alpha - \gamma \rightarrow M_{\hat{r}u}^{\bar{\tau}\uparrow} = \alpha - \gamma + \beta$$

adding e_0 gives

$$M_{\hat{r}u}^{\bar{\tau}\uparrow} = \underbrace{(\alpha + e_0)}_{M_{w\bar{r}}^{\tau\nearrow}} \underbrace{-(\gamma + e_0) + \beta}_{M_{\hat{r}w}} \rightarrow$$

$$M_{\hat{r}w}^{\tau\nearrow} + M_{w\bar{r}}^{\tau\nearrow} = M_{\hat{r}\bar{r}}^{\tau\nearrow}$$

proving that in fact the relationship θ is equality.

2. $\pi_{\hat{\tau}} < \pi_{\tau}$, this means that at $\tau\nearrow$, task $\hat{\tau}$ did not finish its execution and hence clocks $\hat{r}$ and $\hat{w}$ are both active. The analysis for $\hat{r}$ is the same as above; let us see what happens to $\hat{w}$. Figure 5.14 shows the situation graphically.

$$M_{\hat{w}w}^{\tau\nearrow} + M_{w\bar{r}}^{\tau\nearrow} \theta M_{\hat{w}\bar{r}}^{\tau\nearrow}$$

In figure 5.14 we have:

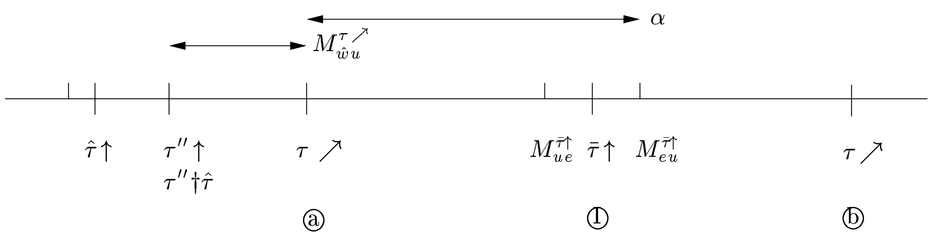

α
$M^{\tau\nearrow}_{\hat{w}u}$
$\hat{\tau}\uparrow$
$\tau''\uparrow$
$\tau''\dagger\hat{\tau}$
$\tau\nearrow$
$M^{\bar{\tau}\uparrow}_{ue}$
$\bar{\tau}\uparrow$
$M^{\bar{\tau}\uparrow}_{eu}$
$\tau\nearrow$
ⓐ
①
ⓑ

Figure 5.14: Case ① $\pi_{\hat{\tau}} < \pi_{\tau}$

- $M^{\tau\nearrow}_{\hat{w}w} = M^{\bar{\tau}\uparrow}_{\hat{w}e} = M^{\tau\nearrow}_{\hat{w}e} = M^{\tau\nearrow}_{\hat{w}u} - e_0$
  The difference between $\hat{w}$ and w at the moment of resuming τ (point ⓐ) is the same as the difference when w was created, point ①, that is at arrival of $\bar{\tau}$; by the property of monotony this difference is kept since both e and $\hat{w}$ were running, that is $\tau\nearrow$ at point ⓐ. Finally, by monotony this value is the same as the difference between the initial value for $\hat{w}$, that is $M^{\tau\nearrow}_{\hat{w}u}$ and e_0.
- $M^{\tau\nearrow}_{w\bar{r}} = M^{\bar{\tau}\uparrow}_{eu} = \alpha + e_0$ and
- $M^{\tau\nearrow}_{\hat{w}\bar{r}} = M^{\bar{\tau}\uparrow}_{\hat{w}u} = M^{\tau\nearrow}_{\hat{w}u} + \alpha$ adding e_0 gives:

$$\underbrace{M^{\tau\nearrow}_{\hat{w}u} + \alpha}_{M^{\tau\nearrow}_{\hat{w}\bar{r}}} = \underbrace{M^{\tau\nearrow}_{\hat{w}u} - e_0}_{M^{\tau\nearrow}_{\hat{w}w}} + \underbrace{\alpha + e_0}_{M^{\tau\nearrow}_{w\bar{r}}}$$

hence proving

$$M^{\tau\nearrow}_{\hat{w}w} + M^{\tau\nearrow}_{w\bar{r}} = M^{\tau\nearrow}_{\hat{w}\bar{r}}$$

Case ②

Recall figure 5.12; we have also two possibilities for $\hat{\tau}$

1. $\pi_{\hat{\tau}} > \pi_{\tau}$, this situation is not possible under our scenario since we are considering τ preempted by $\bar{\tau}$ during its last execution.

2. $\pi_{\hat{\tau}} < \pi_{\tau}$, then $\hat{\tau}$ did not execute at all: its priority being smaller, it must wait at least for τ to finish; only $\hat{r}$ is running. Figure 5.15 shows this situation graphically;

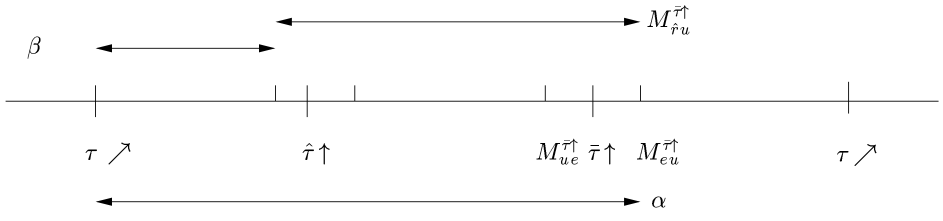

$M^{\bar{\tau}\uparrow}_{\hat{r}u}$
β
$\tau\nearrow$
$\hat{\tau}\uparrow$
$M^{\bar{\tau}\uparrow}_{ue}$
$\bar{\tau}\uparrow$
$M^{\bar{\tau}\uparrow}_{eu}$
$\tau\nearrow$
α

Figure 5.15: Case ② $\pi_{\hat{\tau}} < \pi_{\tau}$

$$M_{\hat{r}w}^{\tau\nearrow} + M_{w\bar{r}}^{\tau\nearrow} \theta M_{\hat{r}\bar{r}}^{\tau\nearrow}$$

- $M_{\hat{r}w}^{\tau\nearrow} = M_{\hat{r}w}^{\bar{\tau}\uparrow} = M_{ue}^{\bar{\tau}\uparrow} = -(\beta + e_0)$
- $M_{w\bar{r}}^{\tau\nearrow} = M_{eu}^{\bar{\tau}\uparrow} = \alpha + e_0$
- $M_{\hat{r}\bar{r}}^{\tau\nearrow} = M_{\hat{r}u}^{\bar{\tau}\uparrow}$
- In figure 5.15 we have:

$$\alpha - \beta = M_{\hat{r}u}^{\bar{\tau}\uparrow}$$

$$\underbrace{(\alpha + e_0)}_{M_{w\bar{r}}^{\tau\nearrow}} - \underbrace{(\beta + e_0)}_{M_{\hat{r}w}^{\tau\nearrow}} = M_{\hat{r}u}^{\bar{\tau}\nearrow}$$

hence proving

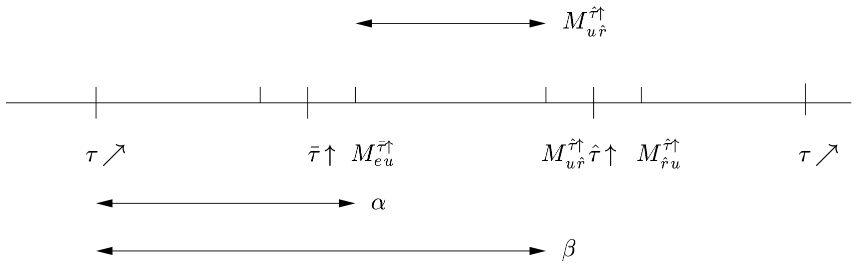
$$M_{\hat{r}w}^{\tau\nearrow} + M_{w\bar{r}}^{\tau\nearrow} = M_{\hat{r}\bar{r}}^{\tau\nearrow}$$

Case ③

Once again, we have two possibilities for $\hat{\tau}$

1. $\pi_{\hat{\tau}} > \pi_\tau$, in this case $\hat{\tau}$ did finish at $\tau^{\nearrow}$ and if $\hat{\tau}$ preempted a task, it should be one with higher priority than τ, so both tasks have finished by the moment $\tau^{\nearrow}$, (point ⓑ) and no clock $\hat{r}$ exists.

2. $\pi_{\hat{\tau}} < \pi_\tau$, then $\hat{\tau}$ did not execute at all, only clock $\hat{r}$ is running. Figure 5.16 shows the scenario.



$M_{u\hat{r}}^{\hat{\tau}\uparrow}$
$\tau\nearrow$
$\bar{\tau}\uparrow$
$M_{eu}^{\bar{\tau}\uparrow}$
$M_{u\hat{r}}^{\hat{\tau}\uparrow}$
$\hat{\tau}\uparrow$
$M_{\hat{r}u}^{\hat{\tau}\uparrow}$
$\tau\nearrow$
α
β

Figure 5.16: Case ③ $\pi_{\hat{\tau}} < \pi_\tau$

$$M_{\hat{r}w}^{\tau\nearrow} + M_{w\bar{r}}^{\tau\nearrow} \theta M_{\hat{r}\bar{r}}^{\tau\nearrow}$$

- $M_{\hat{r}w}^{\tau\nearrow} = M_{\hat{r}w}^{\hat{\tau}\uparrow} = M_{uw}^{\hat{\tau}\uparrow} = -(\beta + e_0)$
- $M_{w\bar{r}}^{\tau\nearrow} = M_{w\bar{r}}^{\hat{\tau}\uparrow} = M_{eu}^{\bar{\tau}\uparrow} = \alpha + e_0$
- $M_{\hat{r}\bar{r}}^{\tau\nearrow} = M_{u\bar{r}}^{\hat{\tau}\uparrow}$
- In figure 5.16 we have:

$$\alpha - \beta = M_{u\hat{r}}^{\hat{\tau}\uparrow}$$

$$\underbrace{(\alpha + e_0)}_{M^{\tau\nearrow}_{w\bar{r}}} \underbrace{-(\beta + e_0)}_{M^{\tau\nearrow}_{\hat{r}w}} = M_{u\hat{r}^{\hat{\tau}\uparrow}}$$

hence proving

$$M^{\tau\nearrow}_{w\bar{r}} + M^{\tau\nearrow}_{\hat{r}w} = M^{\tau\nearrow}_{\hat{r}\bar{r}}$$

Thus, we have proved that the reachability problem in our transition system $(\mathcal{S}, Q, \rightarrow)$ using a decrementation of the form $e = w - \bar{r}$ for preempted tasks, is solvable. Relationships among running clocks can be encoded using a DBM; we have proved that relationships involving stopped clocks when replaced by their differences do not give a difference constraint, but these difference constraints can be deduced from other difference constraints in M, thus they can be eliminated. The following theorem resumes our theory:

Theorem 5.3 *Given a task model as defined in 5.5.1 and a general scheduling policy, the reachability graph of the system can be symbolically characterized using predicates of the form $I \wedge M$ where I is a conjunction of equalities $e = w - \bar{r}$ and M is a* DBM. *Moroever, the reachability graph is finite.*

As a corollary: **the schedulability problem for this class of systems is decidable**.

5.5.4 Properties of the Model

We have shown that the relationships among release clocks for “free” tasks, that is tasks which have not preempted each other, can be implied by the sum of relationships between a preempted task, its already computed time and the corresponding release clocks. This is a very useful property because it reduces the amount of relationships in M. In fact, all those differences envolving released clocks of free tasks can be deduced from a base set of bounds, involving only released clocks from free tasks and w clock from a preempted task and hence matrix representation is also reduced.

The properties of continuity and monotony are exploted for our reachability analysis, implying that it is possible to construct the reachability graph.

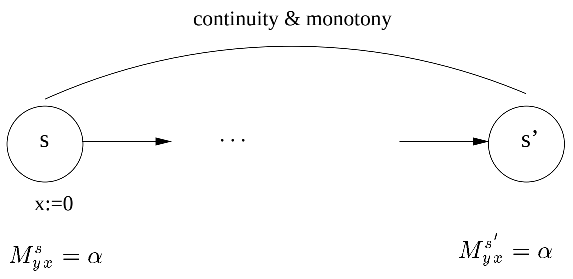


Figure 5.17: Nicety property

In general if we consider two states s and s' in a timed automaton, see figure 5.17, where clock x is reset in s and no reset or update operations are done in between, we can see that the difference is kept; this phenomena is due to the fact that both clocks show a monotonously increasing property (time passing) and also a continuity property (no update is done). Under this context, another clock z (not necessarily continuous) shows the property: $M_{zy} = M_{zx} + M_{xy}$ and hence no need to keep this difference, (intuitively it is as if the stopped time for z were “absorbed” by x and y, both running).

5.6 Final Recipe!

Now that we know the problem can be modelled as the transition system defined in section 5.2, we can sketch an operational approach of our system.

1. Given a RT problem, we can partition it into tasks characterized by timing constraints. If the problem is expressed in Java, we can use techniques such as [28, 32] to cut up the application into smaller tasks.

2. Each task is assigned a fixed or a dynamic priority which is used by the scheduler; naturally, we impose that at arrival of a task, a priority is known.

3. The scheduler keeps a queue Q of tasks, preempted or not, ordered by priority, being the task with highest priority on the head of Q.

4. As a new task τ_i arrives, an admittance test is performed to analyse if its execution leaves the system in a *safe* state, that is, a state where all tasks in Q, including τ_i, finish their executions before their respective deadlines and that no information of preemption is lost. We have given an admittance test for EDF. +

5. If τ_i is accepted:

   - it preempts the currently executing task τ_c if $\pi_i > \pi_c$; we update the information of preemption marking that $p_c := i$ and also setting clock $w_c := e_c$; τ_i joins the queue Q as the new head and τ_c is behind it.
   - it does not preempt τ_c if $\pi_i \leq \pi_c$; in this case, τ_i joins the queue Q somewhere according to its priority.

6. When τ_c finishes, the scheduler can eliminate it from Q but its release clock is kept if $\exists p_j = c$ for some task $\tau_j \in Q$; otherwise all clocks can be eliminated.

7. When a task τ_i resumes execution, its already executed time can be recovered from the difference $e_i := w_i - r_{p_i}$ if $p_i \neq 0$; otherwise $e_i := 0$.

Chapter 6

Conclusions

In this thesis we have followed two main research lines:

- Schedulability of Java-like real time programs
- Decidability of General Preemptive Schedulers

The approach to **schedulability of a Java-like program** is inspired in the use of the synchronization primitives provided by the language to attain good communication among threads and the use of common resources.

Primitives that provide synchronization can have two general forms: a primitive to declare a task is waiting for a response from another task, and conversely a primitive to signal an event to a task. The first primitive is commonly called *wait*, *await*, *receive*, in different languages and even they have different semantics, they do share a feature: the task interrupts its execution and waits until it "hears" a response from another task(s); this event permits the task to awake itself and be ready to resume its execution. The second primitive, commonly referred to as *notify*, *emit*, *send* has as mission awake a task which is (presumably) waiting for this event; in general it is not a blocking operation, that is, the task emiting it continues its execution.

This simple synchronization mechanism permits to implement proper communication among tasks: if a task needs to start as a consequence of an (external) event, then an easy solution is to wait *until* the event happens. We can also use it in a producer/consumer environment where the output of a task is needed as input for another, and we can also use it when some kind of 'order' among tasks is needed to assure functional correctness.

Besides synchronization, tasks may access some common resources (data) in a *competitive* manner, that is, as tasks need data to operate on, they demand them to the data manager who decides about granting or rejecting this demand; tasks do not necessarily communicate each other as in the producer/consumer hypothesis where *cooperation* is explicit, instead they may run independently and the access to resources does not imply some other order to assure functional correctness.

These two facts, inspired us to model a Java like program into *functional components*, that is, pieces of program performing some well defined task; the program is "cut" into two main levels: *application* level and *task* level. The first level spreads a program into major components, called *threads* in our model; each component has its timing constraints and logically performs some general function. The second level spreads a component into minor modules, called *tasks* in our model; these modules may use some (shared) resources and can synchronize with other modules from other threads.

The "cut" of a thread into tasks may be guided by the use of common resources or synchronization primitives. In order to facilitate a *cooperative competition* among tasks, we need to circumscribe the area where a shared resource is used; if a shared resource is used in a piece of code of a thread, then this area can be abstracted as a task. To facilitate synchronization, if a block of code is preceded by an *await* operation, we can abstract it as task.

We have created a graphical and behavioral model of a Java-like real time program, using both synchronization and common resources; we have characterized this model by two timing constraints: *periods* for threads and *execution times* for tasks and also by the set of resources used by each task.

For such a model, we have given a static priority assignment algorithm based on the operations of synchronization; this priorities can be inserted in the Java code to produce a *scheduled Java program.* For shared resources we have given a heuristic technique based on a wait for graph to decide about deadlocks in an off-line manner but this priority assignment could be used in the context of a priority inheritance protocol to assure deadlock freedom during execution. One interesting property of our assignment mechanism is the fact that this order is not complete, that is, tasks involved in synchronization are tied to fixed priorities while independent tasks are free and can be dynamically assigned some convenient priority.

The second axe of this thesis is the problem of **schedulability for preemptive schedulers**; for these schedulers the corresponding computation model is stop watch automaton for which the reachability problem is undecidable, and hence, we could not in general, answer the question "can we reach a final state where all tasks have finished before their deadlines expire?". Even if some results apply to this question, we are constrained by the fact that execution times are bounded but unknown precisely; we have not rely on a worst case execution time hypothesis, but on an interval of execution.

We have created a model of real time tasks characterized by an interval of execution time, based on the idea of *best* and *worst* execution time and also a deadline; periodicity is not a particular restriction of our model; we only need to know a priority for each task, which can be static or dynamic.

For this model, we have presented a transition system where states are characterized by a location and a set of clocks: *release* and *execution* clocks, (as it is classical in these models), and *accumulated execution* clock; besides we have one variable to note preemption. We have characterized this transition system by three event operations: task *arrival*, *resuming* and *completion* and a timed operation:

- Adding one clock per task which counts the accumulated execution time of a task, serves as a mean to let a general preemptive scheduler work correctly.
- This clock is used to *update* the value of the execution clock of a task when it resumes execution after preemption.

We have shown that the reach set of a system of tasks with uncertain but lower and upper bounded execution times, and scheduled according to a preemptive scheduler, can be characterized by constraints involving:

1. time-invariant equations that capture precisely the already executed time of suspended tasks, and
2. a DBM characterizing the differences among the release times of all the active (i.e., suspended or executing) tasks, as long as there exists for each task at most one instance in the system.
3. The structure of the DBM is foreseeable, in the sense that there is a (small) set of basic difference constraints which derive other relationships (not necessarily difference constraints).

This result implies the decidability of the reachability problem for this class of systems. The nice structure of the DBM's generated by the analysis permits a space-efficient implementation, reducing the space needed to represent a DBM from $4n^2$ up to $4n$, in the case of LIFO scheduling policy; moreover, for LIFO, our result does not require introducing any additional clock. For general schedulers, our characterization requires using at most one more clock per task. Actually, the number of additional clocks depends on the number of delayed tasks allowed to be suspended at any time. This number may be controlled via the admission control test. Indeed, the number of extra clocks may be compensated by the more compact representation of the state space.

Besides this, we have given an admittance test for an EDF scheduler; this test is based on deadlines and on blocking times; we have given two bounds for admissibility, taking advantage of our interval of execution: an optimistic (but unsafe) bound which is applicable under the hypothesis of *controllable* execution time or in case of dynamic deadline control; the pesimistic (but safe) bound which is based on the worst case execution time or in case of *uncontrollable* execution time.

The idea of controllable and uncontrollable execution time is useful to characterize some real time applications. Classical real time theory deals with (worst) execution time or uncontrollable execution time, that is, the user or the application itself cannot influence the executing time; but many (modern) application are characterized by the idea of a controllable execution time, that is the application, the environment (and even the user) can influence this time, by given "more or less aproximative results" (for instance, in multimedia, lowering the quality of rendering images); the correctness is not altered by this approximation, and more importantly, it may lead to schedulability when worst execution does not.

The admittance test is thought to help the application to attain schedulability using the execution bounds and controllability.

We have proved that a general scheduler implemented using our method is decidable, in the sense that the schedulability problem expressed as reachability problem is decidable.

6.1 Future Work

We can mention that as future work, we can:

- Give an implementation of our method; indeed, our method is part of a project to create a chain of programs to manage real time applications; starting by a description of the application, its model, the construction of the scheduled program. The implementation must take advantage of the nicety property to create appropriate data structures; then we shold validate it to some applications to prove properties such as liveliness and in general all properties preserved by the reachability graph as mentioned in [17].

- Controllability and uncontrollability of time is not sufficiently exploited in our model, that is, the model does not include controllability of execution time; we use it for the admittance test, but we could design a model based on this idea.

- We base our model on timed automata but we can imagine another base model such as push-down automata. Roughly speaking, the automaton would have arrival, resuming, completion and time passing as operations and the idea is to test the reachability to a final state to deduce schedulability. The stack contains 3-uples of the form (e_i, r_i, w_i) for each active task τ_i.

Bibliography

[1] Yasmina Abdeddaïm and Oded Maler. Job-shop scheduling using timed automata. In Springer Verlag, editor, *Lecture Notes in Computer Science. Special Edition for CAV'2001*, volume 2102, pages 478–492, 2001.

[2] Yasmina Abdeddaïm and Oded Maler. Preemptive job-shop scheduling using stopwatch automata. In Springer Verlag, editor, *Lecture Notes in Computer Science. Special Edition for TACAS 2002*, volume 2280, pages 113–126, 2002.

[3] Advanced Real-Time Systems - Information Society Technologies. *Artist Project: Advanced Real-Time Systems, IST-2001-34820.*

[4] G. Agha. *A model of Concurrent Computation in Distributed Systems.* MIT Press, 1986.

[5] G. Agha. Concurrent object oriented programming. *Communications of the ACM*, 33(9):125–141, 1990.

[6] A.V. Aho, J. E. Hopcroft, and J. D. Ullman. *The design and analysis of computer algorithms.* Addison-Wesley, 1974.

[7] Karine Altisen, Greg Goessler, Amir Pnueli, Joseph Sifakis, Stavros Tripakis, and Sergio Yovine. A framework for scheduler synthesis. In *Proceedings of the 1999 IEEE Real-Time Systems Symposium, RTSS'99*, december 1999.

[8] Karine Altisen, Greg Gossler, and Joseph Sifakis. A methodology for the construction of scheduled systems. In *FTRTFT 2000 Proceedings*, 2000.

[9] Karine Altisen, Greg Gossler, and Joseph Sifakis. Scheduler modeling based on the controller synthesis paradigm. Technical report, Verimag, 2 Av. Vignate, 38610 - Gieres - France, 2000.

[10] R. Alur and D. Dill. Automata for modeling real time systems. *Theoretical Computer Science*, 126(2):183–236, 1994.

[11] T. P. Baker. Stack-based scheduling of real time processes. *The Journal of Real Time Systems*, 3(1):67–100, 1991.

[12] Felice Balarin. Priority assignment for embedded reactive real-time systems. *Languages, Compilers and Tools for Embedded Systems. Workshop LCTES'98*, 1474:146–155, 1998.

[13] Felice Balarin and Alberto Sangiovanni-Vincentelli. Schedule validation for embedded reactive real time systems. In *Proceedings of Design Automation Conference*, Anaheim(CA), 1997.

[14] G. Berry and G. Gonthier. The esterel synchronous programming language: Design, semantics, implementation. *Science of Computer Programming*, 2(19):87–152, 1992.

[15] Greg Bollella. *Real Time Specification for Java*. Addison Wesley, 1999.

[16] Sebastian Bornot, Joseph Sifakis, and Stavros Tripakis. Modeling urgency in timed systmes. In *Lecture Notes in Computer Science. Special Edition for COMPOS'97*, volume 1536, 1998.

[17] Ahmed Bouajjani, Stavros Tripakis, and Sergio Yovine. On-the-fly symbolic model-checking for real-time systems. In *Proc. 18th IEEE Real-Time Systems Symposium, RTSS'97*, San Francisco, USA, December 1997.

[18] Patricia Bouyer, Catherine Dufourd, Emmanuel Fleury, and Antoine Petit. Are timed automata updatable? In *Proceedings of the 12th Int. Conf. on Computer Aided Verification*, pages 464–479, Chicago, USA, July 2000.

[19] Giorgio Buttazzo. Rate monotonic vs edf: Judgment day. In *Proceedings of the 3rd ACM International Conference on Embedded Software (EMSOFT 2003), Philadephia*, October 13-15 2003.

[20] Franck Cassez and Kim Larsen. The impressive power of stopwatches. *Lecture Notes in Computer Science*, 1877:138+, 2000.

[21] Min Chen and Kwei Lin. Dynamic priority ceilings: a concurrency control protocol for real-time systems. *Real Time Systems Journal*, 2(4):325–346, 1990.

[22] D. Dill. Timing assumptions and verification of finite-state concurrent systems. *Proc. 1st Workshop on Computer-Aided Verification. LNCS*, 407, 1989.

[23] Radu Dobrin, Yusuf Ozdemir, and Gerhard Fohler. Task attribute assignment of fixed priority scheduled tasks to reenact off-line schedules. In *Proceeding of RTCSA 2000*, Korea, 2000.

[24] C. Ericsson, A. Wall, and W. Yi. Timed automata as task models for event-driven systems. In IEEE Computer Society Press, editor, *Proceedings of the 6th International Conference on Real Time Computing Systems and Applications*, 1999.

[25] Elena Fersman, Leonid Mokrushin, Paul Pettersson, and Wang Yi. Schedulability anaysis using two clocks. In *ETAPS 2003*, 2003.

[26] Elena Fersman, Paul Pettersson, and Wang Yi. Timed automata with asynchronous processes: Schedulability and decidability. In *ETAPS 2002*, 2002.

[27] Gerhard Fohler. Joint scheduling of distributed complex periodic and hard aperiodic tasks in statically scheduled systems. In *Proceedings of the 16th Real Time Systems Symposium*, Pisa, Italy, 1995.

[28] D. Garbervetsky, C. Nakhli, S. Yovine, and H. Zorgati. Program instrumentation and run-time analysis of scoped memory in java. In *Proceeding of International Workshop on Runtime Verification. ETAPS 2004*, Barcelona. Spain, April 2004.

[29] Thomas A. Henzinger, Peter W. Kopke, Anuj Puri, and Pravin Varaiya. What's decidable about hybrid automata? In *Proceedings of the 27th Annual ACM Symposium on Theory of Computing*, pages 373–382, 1995.

[30] Damir Isovic and Gerhard Fohler. Efficient scheduling of sporadic, aperiodic and periodic tasks with complex constraints. In *Proceedings of the 21st IEEE RTSS*, Florida - USA, november 2000.

[31] Y. Kesten, A. Pnueli, J. Sifakis, and S.Yovine. Integration graphs: A class of decidable hybrid systems. *LNCS. Sepecial Edition on Hybrid Systems*, 736:179–208, 1993.

[32] Christos Kloukinas, Chaker Nakhli, and Sergio Yovine. A methodology and tool support for generating scheduled native code for real-time java applications. In *Proceedings of the Third International Conference on Embedded Software (EMSOFT 2003)*, pages 274–289. Lecture Notes in Computer Science-2855, Springer Verlag, october 2003.

[33] Kim Larsen, Frederik Larsson, Paul Pettersson, and Wang Yi. Efficient verification of real-time systems: compact data structure and state-space reduction. In *Proc. 18th IEEE Real-Time Systems Symposium, RTSS'97*, San Francisco, California, USA, December 1997.

[34] Edward Lee. Embedded software - an agenda for research. *UCB ERL Memorandum M99/63*, December 1999.

[35] Edward Lee and Antonio Snagiovanni-Vincentelli. A framework for comparing models of computation. *IEEE Transactions on CAD*, December 1998.

[36] John Lehoczky and Sandra Thuel. An optimal algorithm for scheduling soft aperiodic tasks in fixed priority preemptive systems. *IEEE Real Time Symposium*, December 1992.

[37] C.L. Liu and James Layland. Scheduling algorithms for multiprogramming in a hard real-time environment. *Journal of the ACM*, 20:46–61, January 1973.

[38] D. C. Luckham and J. Vera. An event-based architecture definition language. *IEEE Transactions on Software Engineering*, 21(9):717–734, September 1995.

[39] Florence Maraninchi. The argos language: Graphical representation of automata and description of reactive systems. In *Proceedings of the IEEE Workshop on Visual Languages*, October 1991.

[40] Jennifer McManis and Pravin Varaiya. Suspension automata: a decidable class of hybrid automata. In *Proc.6th International Conference on Computer Aided Verification, CAV'94, Stanford, California, USA*, volume 818, pages 105–117. Springer-Verlag, 1994.

[41] Jesper Møller. Efficient verification of timed systems using backwards reachability analysis. Technical Report IT-TR-2002-11, Department of Information Technology - Technical University of Denmark, Febrary 2002.

[42] Jesper Møller, Henrik Hulgaard, and Henrik Reif Andersen. Symbolic model checking of timed guarded commands using difference decision diagrams. *Journal of Logic and Algebraic Programming*, 52-53:53–77, 2002.

[43] Jesper Møller, Jakob Lichtenberg, Henrik Reif Andersen, and Henrik Hulgaard. On the symbolic verification of timed systems. Technical Report IT-TR-1999-024, Department of Information Technology - Technical University of Denmark, February 1999.

[44] A.K. Mok. *Fundamental Design Problems for Hard Real Time Environments.* PhD thesis, MIT, 1983.

[45] Xavier Nicollin, Alfredo Olivero, Joseph Sifakis, and Sergio Yovine. An approach to the description and analysis of hybrid systems. *LNCS Special Edition on Hybrid Systems*, 736:149–178, 1993.

[46] Alfredo Olivero. *Modélisation et analyse de systèmes temporisés et hybrides.* PhD thesis, Institut National Polytechnique de Grenoble, France, September 1994.

[47] Amir Pnueli. The temporal logic of programs. In *Proceedings of the 18th Symposium on Foundations of Computer Science, (IEEE FOCS 77)*, 1977.

[48] Ragunthan Rjakumar, Liu Sha, John Lehoczky, and Krithi Ramamritham. An optimal priority inheritance policy for synchronization in real-time systems. In Sang H. Song, editor, *Advances in Real Time Systems*. Prentice-Hall, 1995.

[49] Manas Saksena, A. Ptak, P. Freedman, and P. Rodziewicz. Schedulability analysis for automated implementations of real time object oriented models. In *Proceedings of the IEEE-Real Time Systems Symposium*, 1998.

[50] Lui Sha, Ragunathan Rjakumar, and John Lehoczky. Priority inheritance protocols: An approach to real-time synchronization. *IEEE Transactions on Computers*, 39:1175–1185, 1990.

[51] Joseph Sifakis. Modeling real time systems: Challenges and work directions. In *Lecture Notes in Computer Science. Special Edition for EMSOFT 2001*, volume 2211, 2001.

[52] Joseph Sifakis, Stavros Tripakis, and Sergio Yovine. Building models of real-time systems from application software. *Proceedings of the IEEE, Special issue on modeling and design of embedded systems*, 91(1)::100–111, January 2003.

[53] Joseph Sifakis and Sergio Yovine. Compositional specification of timed systems (extended abstract). In *Proceedings of the 13th Annual Symposium on Theoretical Aspets of Computer Science*, pages 347–359, 1996.

[54] Maryline Silly. The edl server for scheduling periodic and soft aperiodic tasks with resource constraints. *Journal of Time-Critical Computing Systems*, 17:87–111, 1999.

[55] Marco Spuri and Georgio Buttazzo. Efficient aperiodic service under earliest deadline scheduling. In *Proceedings of the IEEE Real Time Systems Symposium*, december 1994.

[56] Marco Spuri and Giorgio Buttazzo. Scheduling aperiodic tasks in dynamic priority systems. *Journal of Real Time Systems*, 10(2), 1996.

[57] Sandra Thuel and John Lehoczky. Algorithms for scheduling hard aperiodic task in fixed priority systems using slack stealing. In *Proceedings of the '94 Real Time Symposium*, Puerto Rico, December 1994.

[58] Ken Tindell. Real time systems and fixed priority scheduling. Technical report, Department of Computer Systems, Uppsala University, 1995.

[59] Sergio Yovine. *Méthodes et outils pour la vérification symbolique de systèmes temporisés.* PhD thesis, Institut National Polytechnique de Grenoble, France, May 1993.

[60] Sergio Yovine. Model-checking timed automata. *Lecture Notes in Computer Science*, 1494, 1998.