

FACULDADE DE ENGENHARIA DA UNIVERSIDADE DO PORTO
DEPARTAMENTO DE ENGENHARIA ELECTROTÉCNICA E DE COMPUTADORES

FORMAL DEVELOPMENT OF SOLUTIONS FOR
REAL-TIME OPERATING SYSTEMS WITH TLA^+ /TLC

JOSÉ MIGUEL SAMPAIO FARIA

Licenciado em Engenharia Mecânica
pela Faculdade de Engenharia da Universidade do Porto

Dissertação submetida para satisfação parcial dos requisitos do grau de
Mestre em Engenharia Electrotécnica e de Computadores

Dissertação realizada sob a orientação do
Professor Doutor Adriano Silva Carvalho
do Departamento de Engenharia Electrotécnica e de Computadores
da Faculdade de Engenharia da Universidade do Porto

PORTO, JANEIRO DE 2008

FORMAL DEVELOPMENT OF SOLUTIONS FOR
REAL-TIME OPERATING SYSTEMS WITH TLA^+ /TLC

Dissertação submetida para satisfação parcial dos requisitos do grau de
Mestre em Engenharia Electrotécnica e de Computadores

À minha família e aos meus bons amigos

Abstract

This thesis addresses the development of solutions for real-time operating systems (RTOS). It specially focuses the realm of problems and challenges characteristic to concurrent systems. The solutions presented are devised recurring to formal methods.

Formal methods are mathematically based techniques for the modeling, specification, development, and verification of computational systems. They are notably valuable helping to produce programs that are free of design errors. In general, different formalisms—languages, methods and tools—are best suited (if not limited) to different applications, depending on multiple criteria, like the stage of the system development cycle and the system nature, i.e., either sequential or concurrent. Because of the nature of the problems ought to analyze, the TLA⁺ formal specification language is adopted. Its associated model checker, TLC, is used to validate the specifications developed.

Many aspects have to be taken in consideration in the development of an RTOS. In this thesis, resource management and communication are the main subjects addressed. Resource management is a key functionality of operating systems. Ensuring the consistency of data in shared resources requires the adoption of resource access protocols. Such a mechanism is devised and formally specified and validated.

Specific scheduling techniques are not addressed. Yet, such techniques are mostly priority-driven. Therefore, the devised resource management mechanism is extended in order to handle priorities. The implications of such exercise are identified and analyzed. The classical priority inheritance protocol is formally specified and a low-level implementation of the protocol is devised. The solution presented allows the calculation at any time of every task's active priority directly from an image of the resource allocation map. It is not necessary to know in which order the system events occurred neither any task's previous priority level. Furthermore, such solution can, under some identified and discussed assumptions, be implemented in a distributed manner.

Much of today's software systems depend on concurrent programming. It is particularly important that RTOSs provide simple and efficient mechanisms for communication between tasks. In this thesis, it is developed a mechanism aiming at fulfilling such requirements. The great novelty of the devised solution is that the kernel is left extraordinarily simple. It is believed that these results could constitute a solid basis for the development of a “nano-kernel” operating

system. The mechanism can be applied to both distributed and non-distributed systems.

All the solutions are presented in the form of TLA^+ specifications and have been validated with the TLC model checker. The combination of the TLA^+ and TLC showed to be an excellent design aid mechanism.

At its essence, this thesis is about reasoning. Rigorous thinking is the best way to make things right. Hopefully, this thesis provides the reader with groundings and methods to increase such capability.

Resumo

Nesta dissertação estuda-se o desenvolvimento de soluções para sistemas operativos de tempo-real, sendo dada particular importância ao conjunto de problemas e desafios característicos dos sistemas concorrentes. As soluções aqui apresentadas são desenvolvidas recorrendo a métodos formais.

Os métodos formais são técnicas baseadas em matemática para a modelação, especificação, desenvolvimento e verificação de sistemas computacionais, podendo ser extremamente úteis no desenvolvimento de sistemas isentos de erros de concepção. Em geral, diferentes formalismos—linguagens, métodos e ferramentas—são melhor indicados (se não limitados) a diferentes aplicações, dependendo de múltiplos critérios, tais como o estágio de desenvolvimento do sistema e a natureza do mesmo, isto é, ou sequencial ou concorrente. Devido à natureza dos sistemas e problemas a analisar ao longo da dissertação, é adoptada a linguagem de especificação formal TLA^+ . Em complemento, é utilizado o *model-checker* TLC para validação das especificações desenvolvidas.

No desenvolvimento de sistemas operativos de tempo-real torna-se necessário considerar múltiplos aspectos. Nesta dissertação, dá-se particular ênfase às temáticas de gestão de recursos partilhados e de comunicação e sincronização entre tarefas. A gestão de recursos partilhados é uma funcionalidade chave dos sistemas operativos. Com efeito, para assegurar a consistência dos dados em recursos partilhados, torna-se necessário definir e adoptar protocolos de acesso aos mesmos. Nesta dissertação, é desenvolvido um mecanismo com tais fins, sendo o mesmo formalmente descrito e validado.

Algoritmos e técnicas específicas de escalonamento não são abordados. No entanto, tendo em consideração que tais técnicas são normalmente baseadas em prioridades, o mecanismo desenvolvido para a gestão de acesso a recursos partilhados é extendido com vista a suportar a gestão de prioridades. As implicações de tal exercício são identificadas e analisadas. É feita uma especificação formal do *priority inheritance protocol* e desenvolvido um mecanismo de implementação do mesmo. A solução apresentada permite o cálculo, a qualquer instante, da prioridade activa de cada tarefa, directamente a partir de uma imagem da alocação de recursos, não sendo necessário saber por que ordem os eventos do sistema ocorreram, nem o nível de prioridade anterior de cada tarefa. Para além do mais, a solução desenvolvida pode, sob algumas condições identificadas e discutidas, ser implementada de uma forma distribuída.

Muito dos sistemas de software actuais dependem de programação concur-

rente. Assim, torna-se particularmente importante que os sistemas operativos de tempo-real forneçam mecanismos simples e eficientes para a comunicação e sincronização entre tarefas. Nesta dissertação, é desenvolvido um mecanismo que visa cumprir tais exigências. Da solução apresentada, destaca-se particularmente o facto de o *kernel* do sistema ser extraordinariamente simples. Com efeito, acredita-se que estes resultados poderiam constituir uma base sólida para o desenvolvimento de sistemas operativos com um “*nano-kernel*”. O mecanismo pode ser utilizado quer em sistemas distribuídos, quer em sistemas monoprocesador.

Todas as soluções desenvolvidas são apresentadas sob a forma de especificações formais em TLA^+ , tendo sido validadas com o *model-checker* TLC. É concluído que a combinação de TLA^+ e TLC constitui uma excelente ajuda para o desenvolvimento de sistemas computacionais.

Na sua essência, esta dissertação dedica-se ao pensar e raciocinar sobre os sistemas a conceber. Na verdade, o pensar de forma rigorosa é a melhor forma de desenvolver sistemas correctamente. É esperado que este texto forneça aos seus leitores um conjunto de bases e ideias para desenvolver tais capacidades.

Résumé

Cette thèse aborde le développement des solutions pour les systèmes d'exploitation temps réel. Elle focalise particulièrement les problèmes et les défis caractéristiques des systèmes concourants. Les solutions présentées sont développées en employant des méthodes formelles.

Les méthodes formelles sont des techniques rigoureuses permettant, à l'aide de logique mathématique, de raisonner, modeler, spécifier, développer, et vérifier des systèmes informatiques. Elles sont notamment valables au développement des programmes qui sont exempts d'erreurs de conception. En général, différents formalismes—langues, méthodes et outils—sont plus adaptés (si non limités) à différentes applications, selon des critères multiples, comme l'étape du cycle de développement et la nature du système, c.-à-d., séquentiels ou concourants. En raison de la nature des problèmes doit analyser, la langue de spécification formelle TLA^+ est adoptée. Son *model-checker* associé, TLC, est utilisé pour valider les spécifications développées.

Pour le développement d'un système d'exploitation temps réel, il faut considérer plusieurs sujets. La gestion de ressources et la communication sont les principaux thèmes abordés en cette thèse. La gestion de ressources est une fonctionnalité primordiale des systèmes d'exploitation. Assurer l'uniformité des données dans les ressources partagées exige l'adoption des protocoles d'accès à ces ressources. Un tel mécanisme est conçu et formellement décrit et validé.

Des techniques spécifiques d'ordonnancement ne sont pas adressées. Cependant, telles techniques sont normalement construites sur l'attribution d'une certaine priorité à chaque tâche. Par conséquent, le mécanisme conçu de gestion de ressource est prolongé afin de manipuler des priorités. Les implications d'un tel exercice sont identifiées et analysées. Le classique *priority inheritance protocol* est formellement spécifié et une implémentation du protocole est conçue. La solution présentée permet à tout moment le calcul des priorités de chaque tâche, directement d'une image de la carte d'attribution de ressource. Il n'est pas nécessaire de savoir dans quel ordre les événements de système se sont produits ni la précédente priorité de chaque tâche. En plus, la solution développée peut, dans de certaines conditions identifiées et discutées, être mis en application d'une façon distribuée.

Beaucoup de systèmes logiciels d'aujourd'hui dépendent de la programmation concourante. Il est particulièrement important que les systèmes d'exploitation temps réel fournissent des mécanismes simples et efficaces pour

la communication entre les tâches. Dans cette thèse, un mécanisme visant remplissant de telles conditions s'est développé. La grande nouveauté de la solution conçue est que le noyau (*kernel*) est laissé extraordinairement simple. On croit que ces résultats pourraient constituer une base pleine pour le développement d'un système d'exploitation de "*nano-kernel*". Le mécanisme développé peut être appliqué aux systèmes distribués et non-distribués.

Toutes les solutions sont présentées sous forme de spécifications écrites en TLA⁺, et ont été validées avec le *model-checker* TLC. La combinaison de TLA⁺ et TLC prouve être un excellent mécanisme d'aide de conception des systèmes.

À son essence, cette thèse est au sujet du raisonnement. La pensée rigoureuse est la meilleure manière de rendre des choses correctes. Cette thèse doit fournir au lecteur des bases et des méthodes pour augmenter telles capacités.

Acknowledgments

This thesis is the culmination of a long journey started some years ago, in Portugal, and that led me to France, Belgium, and, finally, back to Portugal.

First of all, I would like to thank Professor Adriano Carvalho, who always kept track of my work and supervised the write of this thesis.

My first contact with formal methods came from working with Professor Raymond Boute, at the University of Ghent. Prof. Raymond Boute is certainly one of the most clever, sharp, clear thinking individuals on this planet. It was an honor to work close to such a brilliant mind. I'm indubitably grateful for everything I've learned from him. I must also thank my friend and former colleague Hannes Verlinde for all his help, collaboration, and companionship. I thank also my old friend Bruno Oliveira for pushing me into the formal methods community.

Many of the ideas contained in this thesis evolved from my collaboration and discussions with Eric Verhulst, Gjalt G. de Jong, Alexey Nosov, and Valeriy Kamyshniy. I've also learned a lot from them.

Dr. Leslie Lamport is the creator of the TLA⁺ specification language. To him, for that and for many more of his remarkable contributions to the computing science community, I can only express my enormous gratitude and admiration.

Even though not related to the work here presented, my stay in France was a marvellous experience from which I keep the most pleasant memories. I cannot prevent myself from mentioning, first of all, Michel Renier, my friend and supervisor at the time, and also Andy, Mogens, Halina, Célia, Trevor, Ed, Manu, Miguel, Zé, and so many others. It was delightful.

I must also thank Professor Norberto Pires, who gave me the flexibility to find enough time to write this thesis.

At last, I thank all my good friends in Portugal and my family, namely my mother, my father, and the three great women Nadir, Raquel, and Cristina. You are outstanding!

Contents

Abstract	vii
Resumo	ix
Résumé	xi
Acknowledgments	xiii
1 Introduction	1
1.1 Context	1
1.1.1 Motivation: “Ut Faciant Opus Signa”	1
1.1.2 Formal Methods and Dependability	3
1.1.3 Formal Specification and Model Checking	3
1.1.4 The Choice of TLA ⁺ /TLC	5
1.2 Developing Real-Time Systems	6
1.3 Style, Scope, and Contributions	11
1.4 Outline	11
2 Building and Checking a TLA⁺ Specification	13
2.1 Specifying With TLA ⁺	13
2.1.1 What Is TLA ⁺ About	13
2.1.2 How to Build a TLA ⁺ Specification	14
2.2 The TLC Model Checker	19
2.2.1 What to Expect From TLC	19
2.2.2 Model Checking a TLA ⁺ Specification	19
2.2.3 Advices on Using TLC	21
3 Resource Access Management	25
3.1 General Strategy	25
3.2 Modeling Tasks and Resources	26
3.2.1 Task View	26
3.2.2 Resource View	27
3.3 Combining the Models	28
3.3.1 Avoiding Circular Wait Conditions	34
3.3.2 The Resource Access Specification	35

3.4	The Resource Management Mechanism	37
3.4.1	Exclusion of Circular Wait Conditions	41
3.5	Safety and Liveness Checking	41
3.5.1	The Initial Models	43
3.5.2	Machine Closure and TLC's Liveness Checking	43
3.5.3	Proving Implementation	43
3.5.4	A Subtle Difference	45
3.6	Summary	46
4	Priority Handling	47
4.1	Adding Priorities to Tasks	47
4.1.1	Did Fairness Get Killed?	51
4.1.2	The Priority Inversion Phenomenon	52
4.2	The Priority Inheritance Protocol	53
4.3	Intermediate overview	60
4.4	Distributing the Priority Inheritance Computations	60
4.4.1	A Source of Problems	69
4.4.2	Divide to Conquer	71
4.4.3	The Conquest: From Sequential to Combinatorial Calculus	74
4.5	Summary	82
5	Communication and Synchronization	85
5.1	Intertask communication	85
5.1.1	Ports and Tasks Everywhere	86
5.1.2	The TLA ⁺ Specification	86
5.1.3	Constraints for Model Checking	91
5.2	Distributing the System	92
5.2.1	The External Communication Agents	92
5.3	Distributed Communication	97
5.3.1	Avoiding deadlocks	103
5.3.2	Network Considerations	105
5.4	Towards a "nano-kernel"	106
5.5	Summary	106
6	Conclusions	107
	Appendices	111
A	Model-checking Module DistributedInheritance	111
A.1	Configuration File	111
A.2	Error Trace	112

Chapter 1

Introduction

This thesis addresses the subject of formal development of software systems. More precisely, the development of solutions for real-time operating systems using a formal specification language—TLA⁺—and a model checker—TLC. In this chapter, it is presented an introductory overview of the main concepts involved in such an exercise.

We first present the context, surveying the technologies to use, and then (Section 1.2) analyse the issues associated to the development of real-time systems, introducing the specific problems addressed in this thesis. At the end of the chapter, we analyse the contributions of this thesis and present its outline.

1.1 Context

Formal specification and model-checking are techniques that integrate a wider class of activity in the computing science community, generally designated by *formal methods*. Formal methods can greatly contribute to the development of *dependable systems*.

1.1.1 Motivation: “Ut Faciant Opus Signa”

The academic curriculum of any engineer includes the study of classical physics. I can still remember my excitement when my high school physics teacher drew on the blackboard a diagram similar to the one presented in Figure 1.1.¹ The question was: “under which conditions shall the shaded block not fall?”. That drawing fascinated me; I could still picture myself as a kid, trying to do a similar exercise, running with some object close to the chest. Demystifying that childhood challenge was just a step away.

Representing the free body diagram of the block (Figure 1.2), and taking Newton’s second law of motion (the air friction is neglected for simplicity), we get:

¹The informal “I” is exceptionally used here to report a personal experience.

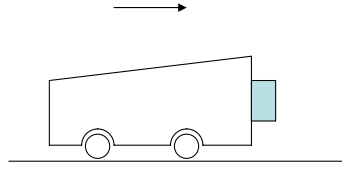


Figure 1.1: System representation

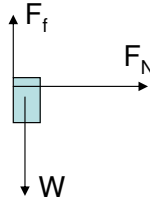


Figure 1.2: Free body diagram

$$\begin{cases} F_f - W = 0 \\ F_N = ma \end{cases} \Leftrightarrow \begin{cases} F_f = mg \\ F_N = ma \end{cases}$$

where W is the weight of the block, F_N is the normal force, and F_f is the force of friction. Knowing that

$$F_f \leq \mu F_N$$

where μ is the coefficient of friction, we get:

$$mg \leq \mu ma$$

which leads us to the conclusion that the block will not fall as long as the acceleration a is such that:

$$a \geq \frac{g}{\mu} \text{ ms}^{-2}$$

As shown, using physic laws and calculation rules, the problem is solved straightforwardly. This simple example illustrates that science and calculation rules can provide significant guidance in problem solving, yielding insight where common intuition is clueless. Such principle is nicely captured in the maxim “*Ut faciant opus signa*” (let the symbols do the work).

The rising question is: “can calculation also be used for deriving software programs?”. The work of Boute [5], for example, provides notable evidence that the answer to the raised question is positive. This thesis attempts to provide insights corroborating such answer.

1.1.2 Formal Methods and Dependability

Dependability is the ability to deliver service that can justifiably be trusted [19]. It is an abstract concept that encompasses a set of attributes such as reliability, availability, safety, integrity and maintainability.

Reliability measures the continuity of correct service. It is expressed by the probability that a piece of equipment or component will perform its required function in the specified manner over a given time period and under stipulated conditions [20]. It is often quantified by MTTF—Mean Time To Failure. *Availability* addresses the readiness for correct service. It is the probability that a system will be functioning correctly at any given time. It is usually quantified by $1 - \text{MTTR}/\text{MTTF}$, where MTTR is the mean time to repair the system and MTTF the mean time to failure. *Safety* regards the absence of catastrophic consequences on the users and the environment. *Integrity* is the absence of improper system alterations. Finally, *maintainability* addresses the ability to undergo modifications and repairs [19].

When a system or component does not perform according to its intended function (for a specified time under specified environmental conditions) we say that we are in the presence of a *failure*. The deviation from correct service may assume different forms that are called service failure modes and are ranked according to failure severities. Since a service is a sequence of the system's external states, a service failure means that at least one external state of the system deviates from the correct service state. This deviation is called an *error*. The adjudged or hypothesized cause of an error is called a *fault* [19].

To attain the attributes of dependability several means exist. These can be grouped in four categories: fault prevention, fault removal, fault tolerance, and fault forecasting. Fault prevention means to prevent the occurrence or introduction of faults. Fault tolerance means to avoid service failures in the presence of faults. Fault removal means to reduce the number and severity of faults. Fault forecasting means to estimate the present number, the future incidence, and the likely consequences of faults.

Fault prevention is precisely the field where formal methods most fit. Fault prevention comprises the set of methods used for attempting to produce programs that are free of design faults. Formal methods are mathematically based techniques for the modeling, specification, development and verification of computational systems (both hardware and software). The same way that in other engineering disciplines, engineers can model and analyze their systems with several mathematical abstractions, formal methods attempt to provide computer engineers with mathematical logic and discrete mathematics as a tool. Formal methods can be notably valuable helping to produce programs that are free of design faults.

1.1.3 Formal Specification and Model Checking

Formal specification languages allow the precise and unambiguous description of systems. Two main approaches to formal systems specification can be distin-

guished, according to the type of language used: logical specification languages and behavioral description languages. The first are based on the use of logics, e.g., predicate calculi, and allow the description of properties and behaviors. Examples of such languages are B, Z, and VDM. Behavioral description languages allow expressing how the system should behave. Their semantics is often expressed in terms of transition systems modulo some abstraction criteria. Examples are SDL, Lotos, Estelle, Esterel, and Petri Nets. Some formalisms like TLA^+ contain elements of both.

In general, formal specification languages combine two great virtues. First, they have precise syntax and semantics, supported by their mathematical rigor. Second, they support abstraction, which facilitates the expression of essential properties and frees the system developer from implementation details and particularities of the target programming language.

TLA^+ is a formalism for describing and reasoning about concurrent systems. The central idea in TLA^+ is that one can use actions—formulas with primed and unprimed variables—in temporal formulas. The first major step in getting beyond traditional programming languages to describe concurrent algorithms was Unity. Unity eliminated the control state, so one just had a single global state to reason about with a single invariant. In TLA^+ , the programming language was eliminated and algorithms are directly written in mathematics. This provides a more powerful and flexible way of describing systems.

Several approaches to program verification have been developed. They are mainly based either on operational reasoning or on axiomatic reasoning, or a combination. Operational reasoning consists of an analysis in terms of the execution sequences of the given program. While this analysis is often successful in the case of sequential programs, it is much less so in the case of concurrent programs. The number of possible execution sequences is often forbiddingly large and it is all too easy to overlook one. Axiomatic reasoning involves the use of logical formulas to characterize sets of states and sets of program behaviors and properties. A program is analyzed by manipulating these logical formulas, rather than by mentally simulating execution.

Automatic strategies have also been developed. Model checking [9, 23] is a notable example. A model checking tool receives a model describing a system and a temporal logic formula specifying the properties required and explores all the possible behaviors of the system. If a property is violated, the model checker returns a state trace illustrating the error, i.e., a counter-example. Model checking is particularly well-suited for the analysis of concurrent systems. It has one drawback: when a verification is performed and no error is found, *total correctness* cannot be guaranteed. It might happen that some relevant property was not specified (and therefore not checked), or, alternatively, that the dimension of the instantiated state space was not large enough to demonstrate the error. Even so, model checking is a remarkably powerful tool, of wide application in the validation of systems. TLC [38] is a model checker for specifications written in TLA^+ .

1.1.4 The Choice of TLA^+ /TLC

In classical engineering disciplines (electrical, mechanical, etc.) tools like Maple and MatLab have a wide scope of utilization and are widely spread and used. This is not the case for tools supporting the use of formal methods in the development of software systems. Instead, there is a rather large set of tools with different and more or less restricted domain of application. Hence, it is particularly important to primarily make an assessment of existing methods and tools and the matching of these with the target application. A shortcoming is the possible subjectivity of such evaluation.

Specification languages can be classified with respect to their prime application domain, i.e., either specialized towards sequential programs, or focusing on concurrent and distributed systems. Well established and representative examples of the first are B, VDM, and Z; whereas CCS, CSP, TLA^+ , Unity and Petri Nets are mainly used to specify and reason about concurrent systems.

The correctness of a system is assessed by confronting the specification of the system/algorithm with the statement of the properties desired and checking whether one implies the other. This can be done by writing proofs by hand or using more or less automated tools. Two main groups of tools can be considered: proof systems and model checkers.

Proof systems generally require some user interoperability. They can check whether a given proof is valid, according to the rules of deduction of the logic concerned, or, given a theorem and its premisses, try to discover a proof such that the theorem follows from the premisses. An hybrid approach can also be utilized: the user proposes large steps and the systems fills in the gaps and takes care of details. Examples of proof systems are Coq, HOL, Isabelle, Nuprl, PVS, and B-related tools.

Model checkers are specially suited for analyzing (descriptions of) concurrent systems. They can differ in many aspects, namely, whether they use branching or linear temporal logic, symbolic or explicit state verification, breadth-first search or depth-first search, real-time or timeless verification, etc..² SMV and NuSMV are symbolic model checkers. Murphi, SPIN and TLC are explicit-state. Kronos and UPPAAL focus on timed systems. Other relevant model checkers are μCRL and FDR. Model checkers more frequently take as input specifically developed languages. Relevant exceptions are FDR, for checking CSP specifications, and TLC (for TLA^+ specifications).

Because tool support is of great importance and since most of the problems ought to be analyzed are related to concurrency issues, model checking is a technique particularly suitable.

Depending on the aforesaid design decisions, model checkers can be more appropriate to verify hardware or software systems. (Naturally, the choice should fall on a model checker primarily intended for software systems.) SMV, NuSMV and Murphi are mainly intended for hardware descriptions verification, which lessens their suitability.

²In Section “The Great Debates” of [11] an interesting discussion over these and other aspects concerning the conception of model checkers can be found.

Kronos and UPPAAL are used to verify real-time systems. However, they take as input rather low-level languages and time constraints are not ought to be addressed. Moreover, logical correctness and timing requirements are orthogonal issues.

SPIN is one of the model checkers of most widespread utilization. Promela is its specifically developed input language. It resembles an imperative language augmented with communication primitives. It is loosely based on CSP. As TLC, SPIN is an explicit state model checker. Therefore, none of the two can handle infinite systems; all the variables have to be bound. TLC and SPIN can both be run in Windows and Unix systems. (SPIN also runs in Mac.) There seems not be a highly significant difference between TLC's and SPIN's model checking capabilities. Their input languages, TLA⁺ and Promela, are, however, enormously different. TLA⁺ is clearly more expressive than Promela. It is a high-level language for the specification of systems. It was not designed with model checking in mind, but as a mean to describe and reason about systems, specially those involving concurrency issues.

In our work, the goal is to develop solutions from scratch, starting with high-level abstractions and gradually refining the solutions obtained. TLA⁺ is especially well suited for such purposes. It most certainly outcomes any possible solution that resembles a programming language.

One other option was to use CSP and its associated model checker FDR. But again, CSP is not as expressive as TLA⁺. CSP is a process algebra, it mainly provides a set of communication constructs. Developing a system based on such constructs might limit and bias the final results.

1.2 Developing Real-Time Systems

An application is considered to be real-time if its correct behavior depends not only on the values produced by the computation, but also on the time at which they are produced [31]. Real-time systems are typically divided into *hard* and *soft* systems, depending on the criticality of satisfying real-time constraints. The real-time constraints of services are materialized by the limit time for their conclusion, called their *deadline*. In hard real-time systems, failure to meet a deadline is considered a system failure and may have severe consequences. Systems in which performance is degraded but not destroyed by failure of a deadline are called soft real-time systems.

The key aspect that distinguishes real-time from general purpose systems is predictability. Whereas, for general purposes, a best-effort approach is normally taken, i.e., it is attempted to execute each service in the least minimum time; in real-time systems the goal is to guarantee that a service is always completed in a pre-determined time.

To support real-time applications, real-time operating systems (RTOS) are developed. The purpose of the RTOS is to support and facilitate the creation of real-time systems. However, the RTOS on its own cannot guarantee that the final result will be real-time; this is dependent on the correct development of

the application software.

A lot of the research on real-time operating systems is centered on scheduling. In fact, scheduling is often considered “the centerpiece of real-time kernels [...] because it is the scheduling algorithm that ensures that tasks meet their deadlines” [31].

The scheduling of tasks in hard real-time systems can either be *static* or *dynamic*. In static approaches, the scheduling is calculated off-line and at runtime the tasks are dispatched according to the precalculated schedule. This can only be done for static systems, where there is a complete prior knowledge of the tasks’ characteristics, and the environment cannot change (or the changes are completely predictable). In dynamic approaches, the scheduling is determined on-the-fly. Therefore, it is possible that new tasks are invoked at unpredictable times. At the cost of higher run-time complexity, dynamic approaches are much more flexible and can easily adapt to changes in the environment.

For dynamic scheduling, a priority based approach is used. Tasks are scheduled on line based on their current priority level. Several algorithms have been developed. These can be divided in two groups, depending on whether tasks are assigned a *fixed priority* or a *dynamic priority*. The rate monotonic (RM) algorithm [21] and the earliest deadline first (EDF) algorithm [21] are well known examples of algorithms for scheduling preemptive periodic tasks.

According to RM, tasks are assigned fixed priorities inversely proportional to their period, so that tasks with the smallest period have the highest priority. According to EDF, priorities are assigned dynamically and are inversely proportional to the absolute deadlines of the tasks, so that the task with the closest deadline has the highest priority.

In general, dynamic priority based protocols have higher CPU utilization bounds. As an example, for periodic tasks whose deadline is equal to their period, EDF has a utilization bound of 100%. That is, EDF can guarantee that all deadlines are met provided that the total CPU utilization is not more than 100%. On the contrary, the RM can only guarantee that all the deadlines are met if the CPU utilization is lower than approximately 69%.³

Traditionally, several characteristics have been pointed out as advantages of fixed priority based algorithms over dynamic ones that counterbalance the lower CPU utilization bound drawback. Among them are, for example, easiness of implementation, easiness of analysis, runtime overhead and predictability in overload conditions. Recent work [7] seriously questions such ideas (described in the cited reference as “misconceptions”), presenting a thorough comparative study between RM and EDF.

For applications that include aperiodic activities to be scheduled together with periodic tasks several algorithms have also been developed. The general approach is to schedule aperiodic request through a periodic server, which allocation a certain amount of time c for aperiodic execution in every period of

³Please note that this does not imply that higher CPU utilization rates cannot be attained. In fact, under certain conditions, a CPU utilization of 100% can be achieved with the RM.

time t (where as the periodic tasks are scheduled according to known algorithms such as RM and EDF). Relevant examples of such algorithms are the Deferrable Server [33] and the Sporadic Server [29], under RM, and the Total Bandwidth Server [30], under EDF.

A common assumption to all the above mentioned algorithms is that, except for the CPU, resource requirements are always taken for granted. This is, it is assumed that a task always has available any resource (memory area, file, shared memory, peripheral device, etc.) that might be necessary for its execution. Naturally, such assumption cannot be guaranteed in practice, which complicates the schedulability analysis of real-time systems, and requires proper support from the RTOS. It is imperative that RTOS provide correct and efficient resource access management mechanisms.

Resources can be said to be either *private* (dedicated exclusively to a particular process) or *shared* (can be used by multiple tasks). To prevent data corruption in shared resources, their access is normally *mutually exclusive*. Mutual exclusion means that a resource can only be held by one process at a time. In the application, the number of shared resources (or intervals of time at which two or more tasks require simultaneous access to the same resource) should be as low as possible, ideally zero. However, that is normally impossible to guarantee.

Developing a mechanism to ensure exclusive access to resources is precisely one of the subjects addressed in this thesis. Apart from the necessary mutual exclusion, the mechanism shall also meet the requirements:

- The **resource** shall be non-preemptable, not the **task**. This means that any task t holding a resource r can have its execution preempted. (What is not allowed is for other tasks than t to execute holding the same resource r before t releases it.)
- The mechanism should be independent from the scheduling policy. This means that it should be possible to “glue” any scheduling policy to it.

Particular attention should be paid to deadlocks. For that, it shall be assured that:

- Circular wait conditions are not allowed. (By circular wait conditions we mean situations where in a group of at least two tasks, each task is waiting for a resource held by another task.)
- The release of resources is not forgotten. Like that, other tasks possibly waiting for the just released resource can lock it and proceed execution.

The development of such a mechanism is presented in Chapter 3 of this thesis.

Despite the fact that independence from the scheduling policy is one of the design goals, resource sharing raises questions that are transversal to most scheduling algorithms. To analyze such questions, extending the resource access

management mechanism developed in Chapter 3 in order to cope with them, is the subject addressed in Chapter 4.

A common problem to priority based scheduling algorithms when using mutually exclusive shared resources is the *priority inversion phenomenon*. Priority inversion occurs when a lower priority task holds a resource that is required by a higher priority task preventing the latter from executing for longer than strictly necessary because some (set of) intermediate priority task (that does not depend on the shared resource) runs in the interim.⁴

To solve the priority inversion phenomenon several specific protocols have been developed. Notable examples are the “priority inheritance protocol” [27] and the “priority ceiling protocol” [27], originally devised for fixed priority based protocols, such as the above mentioned rate monotonic (RM). For dynamic priority based protocols, such as the earliest deadline first (EDF), other solutions such as the “dynamic priority inheritance” [32], the “dynamic priority ceiling” [8], the “stack resource policy” [4], and the “dynamic deadline modification” protocol [12] exist.

The priority ceiling protocol has the advantage over the priority inheritance protocol that it also prevents circular wait conditions. The protocol has, however, a drawback: for an efficient implementation it requires a priori knowledge of all the tasks’ possible resource accesses, which might be extremely difficult, or even impossible. Applying the priority inheritance protocol is, on the contrary, transparent to the application. Such significant advantage leads to the choice of the priority inheritance protocol as an object of study for this thesis. Its formal specification is addressed in Section 4.2.

Many considerations can be made about the usage of both the priority inheritance and the priority ceiling protocols (or others). Above all, the application should avoid the need for their application. Even though both protocols prevent unbounded priority inversion, priority inversion is inherently bad. A high-priority task should never have to wait for a low-priority task that holds a shared resource. This said, it should be noted that such precautions are to be taken by the application developers. Like many other tools, priority inheritance can be dangerous. Nevertheless, correctly implemented and applied, it can highly contribute to the successful development of real-time systems. Independently from the application design, the RTOS has to provide proper support for priority inheritance.

One extra difficulty for implementing the priority inheritance protocol is the possible occurrence of nested resource locks. Add-hoc solutions such as allowing each task to only own one shared resource at a time (which prevents the application complexity from growing too much) are sometimes applied. In the solution to develop herein no such restrictions shall be posed. Except for deadlocking situations, any resource locking map shall be possible to be achieved.

All the solutions reported in this thesis are presented in the form of TLA⁺ specifications. TLA⁺ has the power to allow very high-level descriptions, where

⁴Further details about the priority inversion phenomenon are given in Section 4.1.2 (page 52).

many implementation details are abstracted away. At the same time, TLA⁺ supports refinement, namely through the expression that a lower-level model implements a higher-level description. In Section 4.4 we will devise a low-level implementation of the priority inheritance specification presented in Section 4.2.

There is one aspect that most existing RTOS give less attention: applications are getting more and more distributed and, in such scenario, communication plays a fundamental role. In effect, most of the programming paradigms, programming languages, and software applications in general have been/are developed with centralized sequential execution in mind (even if unconsciously). In a centralized system the processors are located at a single point and the inter-process communication cost is normally negligible compared to the processor execution cost. On the other side, a typical embedded distributed application comprises two communications (input data and output results) for each internal computation, which makes communication not so inexpensive.

In answer to such demand, Chapter 5 describes the development of a mechanism that could serve as the basis of an RTOS with communication (and synchronization) at primary concern. Below, we introduce the central guiding principles of the devised solution:

- *Application target*: The final solution is envisaged to serve embedded applications. Consequently, code size restrictions are to be considered. The system should be as simple and small as possible.
- *Predictability*: A good RTOS should not only provide efficient mechanisms to serve the application developer, but also keep its own time and resource consumptions predictable and accountable. A highly simple and structured kernel is the best way to achieve such requirement.
- *Certification*: There are more and more applications that require certification (such as safety-critical systems for example). Achieving high-level certification can be quite difficult. Simplicity and smallness of the kernel are advantageous from this perspective as well. Like so, only the most elementary services shall be provided in kernel context. As much as possible shall be pushed to task level.
- *Dependability*: Device drivers are the principal failing component in most systems [34]. Consequently, drivers shall be outside the kernel space. In accordance to the previous item, the kernel shall be as concise as possible.
- *Uniformity*. The mechanism for both local and external communications shall be similar. Like that, the location of the resources shall be transparent to the user.

1.3 Style, Scope, and Contributions

This thesis contains several models written in the TLA^+ specification language, being each model accompanied by a textual description. In every description, it is attempted to keep a balance between an intuitive explanation and a rigorous text. On one hand, too much rigour might turn the text considerably harder to understand at a first glance. On the other hand, a too informal description might be too imprecise. It should be understood that the complete rigorous description of the specifications presented are the TLA^+ modules themselves. No spoken language can be fully precise and unambiguous.

The TLA^+ specification language is described in L. Lamport's book [15]. With great honesty, there is not much, if any, that can be learned about TLA^+ from reading this thesis that cannot be learned from the cited reference. Almost everything I know about TLA^+ and concurrency comes from studying Lamport's work. The thesis does provide, however, a large set of examples and advices on how to build, structure, divide, conjoin, and refine TLA^+ modules. Lamport himself claims [13] calling people's attention to Dijkstra's work on self-stabilization to be one of his greatest contributions to computer science. I would be extremely pleased to be able to call some people's attention to Lamport's excellent creation, the TLA^+ specification language.

The solutions developed in this thesis are related to two key aspects in the design of RTOSs, namely, resource management and communication. For both subjects, highly simple and powerful solutions are achieved. The mechanism for resource access management incorporates support for the priority inheritance protocol and presents a conceptually very simple algorithm to calculate every task's priority. For the communication and synchronization between tasks, it is devised an homogeneous solution for both uniprocessor and multiprocessor systems, allowing tasks to interact uniformly no matter their localization.

Globally, it is intended that both the solutions attained and the processes applied to reach them constitute valuable contributions to the interested reader.

1.4 Outline

This thesis is organized in six chapters. Chapter 1 introduces the main concepts that are subject of study in the thesis. Chapter 2 presents the TLA^+ specification language, its associated model checker TLC, and describes the process of building and checking a TLA^+ specification.

Chapter 3 is the first of two chapters dedicated to the development of a resource handling mechanism. In this chapter, a protocol that guarantees exclusive access to shared resources is devised. Such exercise is performed by starting with very simple models and successively refining them. These results are extended in Chapter 4, where priority handling support is added. The implications of adding priorities to tasks and granting the access of tasks to resources in function of their priorities are analyzed. The priority inheritance protocol principles are incorporated and a low level implementation is devised.

Chapter 5 describes a mechanism for communication and synchronization between tasks. The mechanism is first described for local communication (and synchronization) and then extended for distributed applications.

Chapter 6 concludes the thesis by highlighting the most relevant conclusions of the work herein described.

Chapter 2

Building and Checking a TLA⁺ Specification

This chapter presents a concise description of the TLA⁺ language and the TLC model checker. The general process of building and checking a system's TLA⁺ specification is described. The chapter should provide enough support for the reader to understand the TLA⁺ specifications following it.

2.1 Specifying With TLA⁺

TLA⁺ is a specification language especially suited to describing and reasoning about asynchronous, nondeterministic, concurrent systems. It was devised by Leslie Lamport, and is presented in his book [15]. Lamport first developed TLA, the Temporal Logic of Actions [14], and then combined it with classical set-theory and constructs for modularizing large specifications, creating the TLA⁺ specification language.

2.1.1 What Is TLA⁺ About

TLA⁺ is a specification language based on logics. It therefore provides a rigorous, mathematically based mechanism for specifying systems. Precise, rigorous, and unambiguous reasoning is the best way to avoid errors in system's development.

Systems are usually described either with a programming language or with *pseudo-code*. These, however, possess no constructs to describing the properties that the system is supposed to meet. In contrast, TLA⁺ allows the specification of both the system and its desired properties within the same formalism. Correctness of a system means that the formula specifying the implementation implies the formula specifying the desired properties.

Furthermore, while specifying, people normally reason about an abstract algorithm, not about a program that is actually executed. Some aspects of

the system are described and others ignored. A specification is an abstraction. This is naturally done using TLA⁺. The same, however, cannot be said for programming languages.

In conclusion, to specify a system do not use a programming language. Use a specifying language. TLA⁺ is an excellent example. First, its foundation on logics provides a rigorous language, with precise syntax and semantics. Secondly, it supports abstraction, allowing the expression of essential properties and high-level description of behaviors, while ignoring implementation details and particularities of the target programming language. A correct specification is the best start for a correct implementation.

2.1.2 How to Build a TLA⁺ Specification

Specifying a system means characterizing its set of possible behaviors. A behavior is a sequence of states, where a state is an assignment of values to variables. Writing a specification involves abstraction. Some aspects are described and some other ignored. It is essential to choose the proper level of abstraction. This means choosing the variables that represent the system's state and the granularity of the steps that change those variables' values.

The process of writing a specification can be summarized as follows: first, choose the variables and define the initial predicate. Next, write the next-state action as a disjunction of actions describing the different kinds of system operations. Then, write the temporal part and, finally, assert theorems about the specification. Two kinds of properties can be defined: *safety* properties and *liveness* properties. Safety for conditions that may never be violated, and liveness to state that something does eventually happen.

Example

To better illustrate the described methodology, a simple example is presented next. First, an informal specification of the problem is given. Then, a TLA⁺ specification is built, step by step, introducing the different TLA⁺ operators. Finally, in Section 2.2, TLC will be used to check the system.

The problem description is integrally extracted from [26], where a solution written in PROMELA is given.¹ The interested reader may compare the two solutions for a better understanding of the differences between the two languages.

Problem: Four soldiers who are heavily injured, try to flee to their home land. The enemy is chasing them and in the middle of the night they arrive at a bridge that spans a river, which is the border between the two countries at war. The bridge has been damaged and can only carry two soldiers at a time. Furthermore, several land-mines have been placed on the bridge and a torch is needed to sidestep all the mines. The enemy is on their tail, so the soldiers know that they have only 60 minutes to cross the bridge. The soldiers only have a single torch and they are not equally injured. The following table lists the crossing times (one-way!) for each of the soldiers:

¹Recall that PROMELA is the input language of the SPIN model-checker.

soldier S0 5 minutes
 soldier S1 10 minutes
 soldier S2 20 minutes
 soldier S3 25 minutes

Question: Does a schedule exist which gets all four soldiers to the safe side within 60 minutes?

The example is rather simple and different from any program that we may need to construct. Yet, it suffices to illustrate many TLA⁺ aspects. At the same time, similarities with real systems can be identified, such as constrained access to resources (a maximum of two soldiers can be on the bridge at a time), and synchronization needs (a soldier can only cross the bridge in one direction after all the soldiers coming in the opposite direction have arrived).

The first step to build a TLA⁺ specification consists of choosing the variables to represent the state of the system. It is necessary to keep track of the time, the position of every soldier, and coordinate the bridge crossing. Let's use four variables: (1) *timer*, to register the amount of time spent, (2) *unsafe*, representing the set of soldiers that is on the initial side of the bridge at any time, (3) *safe*, which represents the set of soldiers that has crossed the border, and (4) *torch*, to signal when a (set of) soldier(s) can cross the bridge. When *torch* equals "unsafe" the soldiers can cross the bridge from the unsafe to the safe side (and in the opposite direction when it equals "safe").

In TLA⁺, variables are declared with the keyword `VARIABLES`. It is advisable to complement the declaration of the variables with their "type". By "type" it is meant the set of values a variable can assume in any behavior satisfying the specification; TLA⁺ is an untyped language.² The declaration of the (1–4) variables and their "types" stays:

```
VARIABLES timer, unsafe, safe, torch

TypeOk  $\triangleq$ 
   $\wedge$  timer  $\in$  Nat
   $\wedge$  unsafe  $\in$  SUBSET Soldier
   $\wedge$  safe  $\in$  SUBSET Soldier
   $\wedge$  torch  $\in$  {"unsafe", "safe"}
```

The definition of *TypeOk* is optional. It is useful to make the specification easier to understand. In Section 2.2 we will show how to use the definition of *TypeOk* in the verification of the module. The operator `SUBSET S` defines the set of all subsets of *S*. In other words, $T \in \text{SUBSET } S$ iff $T \subseteq S$. The statement $\text{unsafe} \in \text{SUBSET } \text{Soldier}$ formally states that *unsafe* is, at all times, a subset of *Soldier*, where *Soldier* is the set of soldiers.

The set of soldiers *Soldier* is declared as a constant. This is the way in TLA⁺ to leave something unspecified. Indeed, it is not necessary to declare in the TLA⁺ module what are the elements of set *Soldier*. Three more parameters

²A variable *v* is said to have "type" *T* in a specification *Spec* iff $v \in T$ is an invariant of *Spec*. However, this assertion does not imply that variable *v* can only assume values of the set *T*. The reasons to make TLA⁺ an untyped language are addressed in [15] and thoroughly discussed in [18].

are defined: (1) $TimeS(-)$, to represent the time that each soldier needs to cross the bridge, (2) $BridgeMaxLoad$, the maximum number of soldiers that can be on top of the bridge, and (3) $Deadline$, the maximum time available to cross the bridge. All these parameters are then declared as constants:

CONSTANTS $Soldier$, $TimeS(-)$, $BridgeMaxLoad$, $Deadline$

To finalize the first part of the specification, it is also necessary to define the initial state $Init$ of all variables. The following is added to the specification:

$$\begin{aligned} Init \triangleq & \quad \wedge \text{timer} = 0 \\ & \quad \wedge \text{unsafe} = Soldier \\ & \quad \wedge \text{safe} = \{\} \\ & \quad \wedge \text{torch} = \text{"unsafe"} \end{aligned}$$

Next, it is necessary to define the actions that describe the different allowed operations. Two different actions are defined: (1) $UnsafeToSafe(soldset)$, to represent the crossing of the bridge of a set of soldiers, $soldset$, from the unsafe to the safe side, and, conversely, (2) $SafeToUnsafe(soldset)$, the crossing in the other direction. Their definition is shown bellow. The symbols $\wedge$, $\vee$, $\exists$, $\cup$, and $\in$ have their traditional mathematical meaning.

$$\begin{aligned} HighestTime(soldset) & \triangleq \\ & \quad \text{LET } slowest \triangleq \text{CHOOSE } i \in soldset : \forall j \in soldset : TimeS(i) \geq TimeS(j) \\ & \quad \text{IN } TimeS(slowest) \\ \\ UnsafeToSafe(soldset) & \triangleq \\ & \quad \wedge \text{torch} = \text{"unsafe"} \\ & \quad \wedge \text{unsafe}' = \text{unsafe} \setminus soldset \\ & \quad \wedge \text{safe}' = \text{safe} \cup soldset \\ & \quad \wedge \text{timer}' = \text{timer} + HighestTime(soldset) \\ & \quad \wedge \text{torch}' = \text{"safe"} \\ \\ SafeToUnsafe(soldset) & \triangleq \\ & \quad \wedge \text{torch} = \text{"safe"} \\ & \quad \wedge \text{safe}' = \text{safe} \setminus soldset \\ & \quad \wedge \text{unsafe}' = \text{unsafe} \cup soldset \\ & \quad \wedge \text{timer}' = \text{timer} + HighestTime(soldset) \\ & \quad \wedge \text{torch}' = \text{"unsafe"} \end{aligned}$$

An action is defined as a conjunction of guarding conditions, some possible internal computations, and actualizations of variables. The guarding condition of $UnsafeToSafe(soldset)$ is that $\text{torch} = \text{"unsafe"}$. The variables actualizations are: unsafe' , the new value of unsafe , equals its old value, unsafe without the set of soldiers $soldset$. Hence, $\text{unsafe}' = \text{unsafe} \setminus soldset$. The symbol $\setminus$ is the TLA⁺ operator for set difference. The set $S \setminus T$ is the set of elements of S that are not in T . The ordering of the conjuncts is a matter of taste. Every action is atomic.

For the actualization of timer it is necessary to determine highest value of the times spent to cross the bridge from the soldiers in $soldset$. The function $HighestTime(soldset)$ makes this "calculation". Its definition introduces a new

TLA^+ operator, the **CHOOSE** operator. The expression $\text{CHOOSE } v \in S : e$ equals a value v from the set S that satisfies expression e .

The **LET/IN** construct is also introduced. The **LET** clause consists of a sequence of definitions whose scope extends until the end of the **IN** clause. A definition in a **LET** is just like an ordinary definition in a module, it is normally used to make an expression easier to read. Using a **LET** to eliminate common subexpressions can often significantly shorten and simplify an expression. In the definition of *HighestTime*, *slowest* is defined to equal the soldier s in *soldset* that has the highest value of $\text{TimeS}(s)$, i.e., the soldier that takes longer to cross the bridge.

Note that, in the definition of the actions, nothing is said about *soldset*. The actions definition is totally general. Constraints shall be added in the definition of the next-state relation *Next*.

If we try to solve the problem mentally, an obvious hint is that each time two soldiers (the maximum allowed) cross the bridge to the safe side, only one soldier will come back, carrying the torch. However, this constitutes already part of the solution to the problem. Defining that would limit the set of possible behaviors. We leave the TLA^+ specification as general as possible, by stating that the number of soldiers that crosses the bridge in any direction has to be between one and *BridgeMaxLoad*:

$$(\text{Cardinality}(\text{soldset}) \geq 1) \wedge (\text{Cardinality}(\text{soldset}) \leq \text{BridgeMaxLoad})$$

The operator *Cardinality* is predefined in the *FiniteSets* module.³ The TLA^+ tools come with a set of predefined modules, namely *Naturals*, *Integers*, *Reals*, *Sequences*, *FiniteSets*, *Bags*, *RealTime*, and *TLC*. The definitions there contained can be incorporated into any other module with the operator **EXTENDS**. The **EXTENDS** statements should be placed at the beginning of the module. Our specification shall contain:

EXTENDS *Naturals* , *FiniteSets*

The *Naturals* module is also incorporated because it contains the definition of *Nat* (used in *TypeOk*), and other operators like $+$, $\geq$, and $\leq$.

The next-state relation *Next* is defined as a disjunction of all the actions:

$$\begin{aligned} \text{Next} \triangleq & \vee \exists \text{soldset} \in \text{SUBSET}(\text{unsafe}) : \\ & \wedge \text{Cardinality}(\text{soldset}) \geq 1 \wedge \text{Cardinality}(\text{soldset}) \leq \text{BridgeMaxLoad} \\ & \wedge \text{UnsafeToSafe}(\text{soldset}) \\ & \vee \exists \text{soldset} \in \text{SUBSET}(\text{safe}) : \\ & \wedge \text{Cardinality}(\text{soldset}) \geq 1 \wedge \text{Cardinality}(\text{soldset}) \leq \text{BridgeMaxLoad} \\ & \wedge \text{SafeToUnsafe}(\text{soldset}) \end{aligned}$$

To finalize the specification, some liveness conditions can be added. We will add the expression:

$$\text{WF}_{\text{vars}}(\text{Next})$$

³Recall that the cardinality of a finite set S is the number of elements of S .

In general, $WF_v(A)$ asserts that, if A ever becomes forever enabled, then an A step must eventually occur (WF stands for *Weak Fairness*). If we have two actions, A and B , and we want to specify weak fairness of both, we can write: $WF_v(A) \wedge WF_v(B)$.

However, under certain conditions, this expression can be simplified to: $WF_v(A \vee B)$. The conditions under which $WF_v(A \vee B) \equiv WF_v(A) \wedge WF_v(B)$ are two. First, (1) whenever A is enabled, B can never become enabled unless an A step eventually occurs. Conversely, (2) whenever B is enabled, A can never become enabled unless a B step eventually occurs. In our example, these conditions are met, being A and B the actions *UnsafeToSafe* and *SafeToUnsafe*. Therefore, the liveness expression can be simplified to $WF_{vars}(Next)$.

The specification *Spec* of our problem is then summarized in the formula:

$$Spec \triangleq Init \wedge \Box[Next]_{vars} \wedge WF_{vars}(Next)$$

The formula asserts about a behavior that (a) its initial state satisfies *Init*, and (b) each of its steps satisfies *Next*. To express (b), the temporal operator $\Box$ (pronounced *box*) is used. The temporal formula $\Box F$ asserts that formula F is always true. In our case, $\Box Next$ is the assertion that *Next* is true for every step in the behavior. Therefore, $Init \wedge \Box Next$ is true of a behavior iff the initial state satisfies *Init* and every step satisfies *Next*. The formula should, however, also allow steps that leave all the variables unchanged, called *stuttering steps*. So, taking *vars* as the tuple of all the variables, $\Box Next$ should be replaced by $\Box(Next \vee (vars' = vars))$. In TLA⁺, this is abbreviated as $\Box[Next]_{vars}$.

At last, at the bottom of the specification, we can assert theorems. For that, let's go back to the crux of our problem. The question was whether a schedule exists that gets all four soldiers to the safe side within 60 minutes. To leave the specification general, we drop the 60 minutes and use parameter *Deadline* and define the property *SoldiersDeath*:

$$SoldiersDeath \triangleq \Box(Cardinality(unsafe) = 0 \Rightarrow timer > Deadline)$$

The formula asserts that for all the correct behaviors of the system, in every state where $Cardinality(unsafe) = 0$, *timer* is strictly greater than *Deadline*. More informally, whichever procedure taken by the soldiers to cross the bridge takes longer than *Deadline*. Next, in Section 2.2, it is shown how TLC checks whether property *SoldiersDeath* is true of specification *Spec* or not.

The full TLA⁺ specification is put into module *Soldiers*, shown in Figure 2.1. From the description above, reading the module should be straightforward. It starts with

MODULE *Soldiers*

which begins a module named *Soldiers*. It then contains all the described definitions in a pre-defined order, and is terminated by the symbol

The intermediate lines

can appear anywhere between statements. They are purely cosmetic and have no meaning.

Further aspects about the TLA⁺ specification language will be introduced throughout the text when appropriate.

2.2 The TLC Model Checker

TLC is an on-the-fly explicit-state model checker for specifications written in TLA⁺. It was developed by Yuan Yu, and first presented in [38]. It is available through the TLA⁺ web page.

2.2.1 What to Expect From TLC

TLC model checks TLA⁺ specifications. TLA⁺ was, however, not designed with model checking in mind. This distinguishes TLC from most of the other model checkers, which take as input specifications written in low-level specifically developed languages. Because TLA⁺ is such a high-level language, not every possible specification can be model checked. Still, TLC can check a large subclass of TLA⁺ specifications, like the ones present in this document.

Because of TLA⁺ expressiveness, TLC *executes* a specification interpretively. Like that, it can be significantly slower than conventional model checkers for specifying simple systems. For large specifications, TLC's performance has been reported as comparable [16]. To handle performance limitations, most verifications must be made with small instances. This is, however, a general problem of the model checking technique, transverse to most tools.

TLC can make use of shared-memory multiprocessing and networked computing. It can almost obtain a factor of n speedup by using n processors, for n up to at least 8 [17]. It can therefore take advantage of the coming generations of multicore computers. In [17] it is reported an example where a 40 hours uniprocessor execution was reduced to less than an hour on a first-generation 384-processor Azul Systems computer [35].

TLC constitutes, at the same time, a great tool to rapidly get acquainted and gain proficiency in writing TLA⁺ specifications. The virtues of using a tool to learn a language like TLA⁺ are highly significant.

2.2.2 Model Checking a TLA⁺ Specification

To run TLC, two different inputs are needed: (1) a TLA⁺ module, and (2) a *configuration file*. The TLA⁺ module can naturally incorporate any number of other modules. The configuration file must contain the definition of all the parameters of the specification. It is also necessary to declare the name of the specification to be checked, and the list of properties and invariants. Figure 2.2 shows the configuration file used to check the *Soldiers* module.

MODULE <i>Soldiers</i> EXTENDS <i>Naturals</i>, <i>FiniteSets</i> CONSTANTS <i>Soldier</i>, <i>TimeS</i>(_), <i>BridgeMaxLoad</i>, <i>Deadline</i> VARIABLES <i>timer</i>, <i>unsafe</i>, <i>safe</i>, <i>torch</i> <i>vars</i> $\triangleq$ $\langle timer, unsafe, safe, torch \rangle$ <i>TypeOk</i> $\triangleq$ $\wedge timer \in Nat$ $\wedge unsafe \in \text{SUBSET } Soldier$ $\wedge safe \in \text{SUBSET } Soldier$ $\wedge torch \in \{ "unsafe", "safe" \}$ <i>Init</i> $\triangleq$ $\wedge timer = 0$ $\wedge unsafe = Soldier$ $\wedge safe = \{ \}$ $\wedge torch = "unsafe"$ <i>HighestTime</i>(<i>soldset</i>) $\triangleq$ LET <i>slowest</i> $\triangleq$ CHOOSE $i \in soldset : \forall j \in soldset : TimeS(i) \geq TimeS(j)$ IN <i>TimeS</i>(<i>slowest</i>) <i>UnsafeToSafe</i>(<i>soldset</i>) $\triangleq$ $\wedge torch = "unsafe"$ $\wedge unsafe' = unsafe \setminus soldset$ $\wedge safe' = safe \cup soldset$ $\wedge timer' = timer + HighestTime(soldset)$ $\wedge torch' = "safe"$ <i>SafeToUnsafe</i>(<i>soldset</i>) $\triangleq$ $\wedge torch = "safe"$ $\wedge safe' = safe \setminus soldset$ $\wedge unsafe' = unsafe \cup soldset$ $\wedge timer' = timer + HighestTime(soldset)$ $\wedge torch' = "unsafe"$ <i>Next</i> $\triangleq$ $\vee \exists soldset \in \text{SUBSET } (unsafe) :$ $\wedge Cardinality(soldset) \geq 1 \wedge Cardinality(soldset) \leq BridgeMaxLoad$ $\wedge UnsafeToSafe(soldset)$ $\vee \exists soldset \in \text{SUBSET } (safe) :$ $\wedge Cardinality(soldset) \geq 1 \wedge Cardinality(soldset) \leq BridgeMaxLoad$ $\wedge SafeToUnsafe(soldset)$ <i>Spec</i> $\triangleq$ <i>Init</i> $\wedge \Box [Next]_{vars} \wedge WF_{vars}(Next)$ <i>SoldiersDeath</i> $\triangleq$ $\Box (Cardinality(unsafe) = 0 \Rightarrow timer > Deadline)$ THEOREM <i>Spec</i> $\Rightarrow \Box TypeOk$ THEOREM <i>Spec</i> $\Rightarrow SoldiersDeath$

Figure 2.1: The *Soldiers* module.

```

(* ..... *)
(*          CONFIG Soldiers.cfg          *)
(* ..... *)

SPECIFICATION  Spec

CONSTANTS      Soldier = {s0,s1,s2,s3}
                TimeS(s0) = 5
                TimeS(s1) = 10
                TimeS(s2) = 20
                TimeS(s3) = 25
                BridgeMaxLoad = 2
                Deadline = 60

INVARIANT      TypeOk

PROPERTY       SoldiersDeath

(* ..... *)

```

Figure 2.2: Configuration file for module *Soldiers*.

Invariants could be defined and checked as properties. For example, we could make the definition $TypeOkProp \triangleq \Box TypeOk$, and replace the invariant declaration by `PROPERTY TypeOkProp`. The `INVARIANT` statement is just a simplification to allow the omission of the temporal operator $\Box$.

TLC can be run in two different modes: *model checking* and *simulation*. The default mode is model checking, in which it tries to find all the reachable states. In simulation mode, TLC randomly generates behaviors of a fixed maximum length.

Running TLC with our *Soldiers* module and the configuration file, it rapidly finds a behaviors that violates property *SoldiersDeath*. This means that the solvers are save, the bridge can be crossed within 60 minutes. TLC prints an error trace illustrating the behavior that violated the property, called the *counter example*. The output of TLC is shown in Figure 2.3. The reader has probably solved the problem mentally by now; anyhow the error trace is fairly readable.

To try different scenarios, i.e., changing, for example, the number of soldiers and/or the deadline, all the necessary modifications can be made in the configuration file. The TLA^+ specification does not need to be changed. Naturally, specifications should always be made like that.

2.2.3 Advices on Using TLC

Several options can be chosen to run TLC. There is one that should always be enabled, namely the `-coverage num` option. This option causes TLC to

```

TLC Version 2.0 of January 16, 2006
Model-checking
Parsing file Soldiers.tla
[...]
Finished computing initial states: 1 distinct state generated.
Error: Invariant SoldiersDeath is violated.
The behavior up to this point is:

STATE 1: <Initial predicate>
^ safe = {}
^ torch = "unsafe"
^ timer = 0
^ unsafe = {s0, s1, s2, s3}

STATE 2: <Action line 37, col 12 to line 39, col 81 of module Soldiers>
^ safe = {s0, s1}
^ torch = "safe"
^ timer = 10
^ unsafe = {s2, s3}

STATE 3: <Action line 41, col 12 to line 43, col 81 of module Soldiers>
^ safe = {s1}
^ torch = "unsafe"
^ timer = 15
^ unsafe = {s0, s2, s3}

STATE 4: <Action line 37, col 12 to line 39, col 81 of module Soldiers>
^ safe = {s1, s2, s3}
^ torch = "safe"
^ timer = 40
^ unsafe = {s0}

STATE 5: <Action line 41, col 12 to line 43, col 81 of module Soldiers>
^ safe = {s2, s3}
^ torch = "unsafe"
^ timer = 50
^ unsafe = {s0, s1}

STATE 6: <Action line 37, col 12 to line 39, col 81 of module Soldiers>
^ safe = {s0, s1, s2, s3}
^ torch = "safe"
^ timer = 60
^ unsafe = {}

571 states generated, 277 distinct states found, 148 states left on queue.
The depth of the complete state graph search is 6.

```

Figure 2.3: TLC's output model checking module *Soldiers*.

print, every *num* minutes, the number of times every action conjunct has been used in constructing a new state. The magnitude of the values printed can provide useful hints. Particularly, a value of 0 indicates that the conjunct was never “executed”, which might indicate an error in the specification. This is specially useful if none of the specified properties can illustrate that error, which may often happen, specially in the initial stages, where the temptation to keep “building” the model and leaving the specification of desired properties for latter is higher.

State explosion *is* a problem. The first TLC runs should always be made with small instances. Very easily a system may need several hours or days to be model checked. However, many mistakes might be caught with tiny instances, which can only take a fraction of a second to model check. It is a good principle to use a small instance to correct most of the problems and then keep growing it looking for new errors. It can be that a specification has an error that would take such a large instance to be detected that it cannot be model checked in a feasible time. This is a limitation of model checking. TLC can be run in simulation mode on large models. It is certainly not effective, but with luck the error may be found.

The fact that TLC finds no error in a specification should be faced with extreme caution. In many cases that does not mean that the model is “correct”. It can be that the properties specified may not be enough to illustrate an error. Or, as mentioned above, that the size of the state space is not enough to illustrate it. In other cases, it can be that what is written does not really match what is intended, and what we mentally assume as “correct” is not exactly what is being checked. This is a problem that can be gradually solved with experience. Experience will help one gain confidence of the correspondence between mental expectations and/or assumptions and the formal TLA⁺ specification.

TLC does not preserve the exact semantics of the operators CHOOSE and CASE. In practice, that violation is normally not important when using the CHOOSE operator on its own. For CASE statements, it becomes important when the different “branches” are not mutually disjoint. TLC will only behave correctly if they do so. For situations when one wants to specify not mutually disjoint branches, it is better to recur to the common $\wedge$ and $\vee$ operators.⁴

⁴The module *SimpleResourceManagement*, shown in Section 3.4, on page 37, contains an example where the use of CASE would be erroneously checked by TLC.

Chapter 3

Resource Access Management

As introduced in Section 1.2, resource management is a key functionality of operating systems. A *resource* is any software structure that can be used by a process to advance its execution. For example, it can be a data structure, a set of variables, a memory area, a file, or a set of registers of a peripheral device. Resources can be said to be either *private* or *shared*. A private resource is dedicated exclusively to a particular process; a shared resource can be used by multiple tasks. To ensure the consistency of data in shared resources, their access is normally *mutually exclusive*. Mutual exclusion means that a resource can only be held by one process at a time. Like so, operating systems must adopt appropriate resource access protocols. This chapter illustrates the development of such a mechanism with TLA⁺/TLC.

3.1 General Strategy

This is the first chapter of this thesis dedicated to the development of solutions for real-time operating systems. Such task is extended up to Chapter 5. Throughout the chapters, several problems regarding the design of RTOS are addressed. The general strategy for model building is uniform. Initially, simpler models are built. These models are then merged together, and additional detail and complexity is added to them. The relations between the models are explicitly expressed in TLA⁺. Building models in such way combines both composition and refinement [1, 37]. This is an excellent method to develop large specifications. Large models are built in a structured way and many subtle details are easily detected and solved.

In this chapter, a mechanism to ensure exclusive access of tasks to resources is developed. The models developed in this chapter will be used as the “simpler” models of the ones developed in Chapter 4.

3.2 Modeling Tasks and Resources

A *task* designates a unit of execution. Traditionally, several different terms are used such as *process*, *thread*, *task*, or *job*. In many cases differences are defined between them. For example, a process can be said to be splittable into multiple threads. No such distinctions are made here. Every running entity shall be called a task. Each task is uniquely identified.

3.2.1 Task View

A task ta that needs to use a resource must first grant the access to it. If the resource is being used by another task tb , ta must wait until tb releases the resource. Without further details on this mechanism, it is clear that there will be times when the task execution is blocked. Here, the task is said to be *waiting*. When a task is not waiting, it can either be *running* or *ready*, depending on the scheduling policy. Figure 3.1 illustrates the state transition diagram of a task. A formal TLA⁺ specification of such a system is module *TaskView*, shown in Figure 3.2.

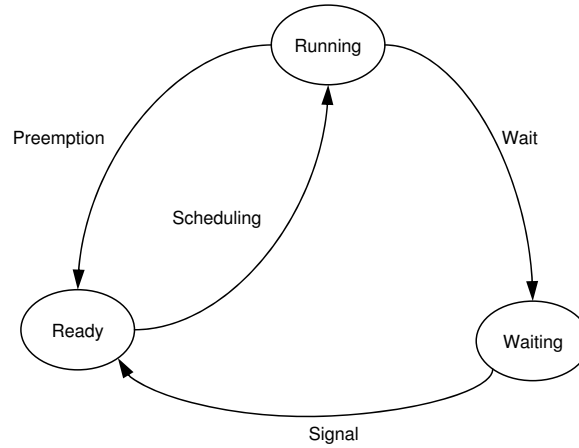


Figure 3.1: Task state transition diagram.

In TLA⁺, a state is an assignment of values to variables. Therefore, variable *TaskState* is used. It is formally defined as a function that maps every task to one of three values: “Ready”, “Running”, or “Waiting”. The *Task* parameter, declared in the CONSTANTS statement, is the set of all tasks.

The actions *TVScheduling*, *TVPreemption*, *TVWait*, and *TVSignal* represent every possible change of “state” of a task.

This model introduces the EXCEPT statement. For any function f , the expression $[f \text{ EXCEPT } ![c] = e]$ is the function $\hat{f}$ that is the same as f except with

$\hat{f}[c] = e$. This function can also be written as

$$[x \in \text{DOMAIN } f \mapsto \text{IF } x = c \text{ THEN } e \text{ ELSE } f[x]]$$

but the first is more concise and more readable.

Note that the model mentions nothing about resources. This is totally abstracted away. It shall be introduced later.

```

1  ┌────────────────────────── MODULE TaskView ───────────────────┐
2  CONSTANTS Task
3
4  VARIABLES TaskState
5
6  TVTypeOk  $\triangleq$  TaskState  $\in$  [Task  $\rightarrow$  {"Ready", "Running", "Waiting"}]
7
8  TVInit  $\triangleq$  TaskState = [t  $\in$  Task  $\mapsto$  "Ready"]
9  ┌────────────────────────────────────────────────────────────────┐
10 TVScheduling(t)  $\triangleq$ 
11    $\wedge$  TaskState[t] = "Ready"
12    $\wedge$  TaskState' = [TaskState EXCEPT ![t] = "Running"]
13
14 TVPreemption(t)  $\triangleq$ 
15    $\wedge$  TaskState[t] = "Running"
16    $\wedge$  TaskState' = [TaskState EXCEPT ![t] = "Ready"]
17
18 TVWait(t)  $\triangleq$ 
19    $\wedge$  TaskState[t] = "Running"
20    $\wedge$  TaskState' = [TaskState EXCEPT ![t] = "Waiting"]
21
22 TVSignal(t)  $\triangleq$ 
23    $\wedge$  TaskState[t] = "Waiting"
24    $\wedge$  TaskState' = [TaskState EXCEPT ![t] = "Ready"]
25 ┌────────────────────────────────────────────────────────────────┐
26 TVNext  $\triangleq$ 
27    $\exists t \in$  Task :
28      $\vee$  TVScheduling(t)
29      $\vee$  TVPreemption(t)
30      $\vee$  TVWait(t)
31      $\vee$  TVSignal(t)
32
33 TVSpec  $\triangleq$  TVInit  $\wedge \square$ [TVNext]TaskState
34 ┌────────────────────────────────────────────────────────────────┐
35 THEOREM TVSpec  $\Rightarrow \square$ TVTypeOk
36 └────────────────────────────────────────────────────────────────┘

```

Figure 3.2: The *TaskView* module.

3.2.2 Resource View

Let's now introduce a new model, independent from the *TaskView* module. It shall represent the state of a resource at every time. For that, variable *ResState* is used. Its TLA⁺ definition is

$$ResState \in [Resource \rightarrow [Locked : \text{BOOLEAN} ,$$

$$\begin{aligned} WL &: \text{SUBSET } (Task), \\ Owner &: Task \cup \{NoTask\} \end{aligned}$$

Variable *ResState* is defined as a function that maps every resource to a record of three fields: *Locked*, *WL*, and *Owner*. When a resource *r* is being used by a task *ta*, the field *Locked* is set to TRUE and the identifier of the task *ta* is written in the *Owner* field. The (possibly empty) set of tasks waiting on a resource is registered in the resource's *waiting list*, field *WL*.

The value *NoTask* that may be assigned to the *Owner* field represents any arbitrary value that does not belong to the *Task* set. Its formal definition in TLA^+ is

$$NoTask \triangleq \text{CHOOSE } t : t \notin Task$$

However, TLC cannot handle such an unbounded operation. Therefore, *NoTask* is simply defined as a parameter of the specification.¹

The specification appears in module *ResourceView*, in Figure 3.3. The actions defined represent the different ways through which a resource state can change.

Action *RVTaskLocksResource* represents the locking of a resource by a task. A task *t* may lock a resource *r* if *r* is not yet locked and if *t* is not already waiting for resource *r*.

Action *RVTaskReleasesResource* represents the moment at which a task *t* owning a resource *r* ceases its utilization. The resource's *Locked* field is set to FALSE, and the *Owner* is set to *NoTask*.

A task *t* that neither owns resource *r* nor waits for it, can at any time be added to its waiting list. This is represented by action *RVTaskGoesToWL*.

Finally, when a resource *r* is not locked, any task *t* that belongs to *r*'s waiting list can lock the resource. The value of *ResState* is updated such that task *t* is registered as the owner of the resource and leaves the resource's waiting list. Action *RVTaskGoesFromWLtoOwnership* represents the describe scenario.

Figure 3.4 depicts the described behavior. It does not illustrate the resource's waiting lists.

3.3 Combining the Models

To develop a resource access mechanism, we can now combine the modules *TaskView* and *ResourceView*. The resulting module is called *InternalResourceAccess*, and is shown in Figure 3.6.

The new specification contains three variables: (i) *TaskState*, from the *TaskView* module, (ii) *ResState*, from the *ResourceView* module, and (iii) *TaskOw*.

¹In rigour, the definition of *NoTask* could be left in the model as the unbounded choice and TLC could overwrite it.

```

1  ┌────────────────────────── MODULE ResourceView ───────────────────┐
2  │ CONSTANTS Task, Resource, NoTask                                  │
3  │                                                                    │
4  │ VARIABLES ResState                                              │
5  │                                                                    │
6  │ RVTypeOk  $\triangleq$  ResState  $\in$  [Resource  $\rightarrow$  [Locked : BOOLEAN ,
7  │                                                                    │
8  │                                                                    │
9  │                                                                    │
10 │ RVInit  $\triangleq$  ResState = [r  $\in$  Resource  $\mapsto$  [Locked  $\mapsto$  FALSE,
11 │                                                                    │
12 │                                                                    │
13 │                                                                    │
14 │ RVTasksLocksResource(t, r)  $\triangleq$ 
15 │    $\wedge t \notin \text{ResState}[r].WL$ 
16 │    $\wedge \text{ResState}[r].Locked = \text{FALSE}$ 
17 │    $\wedge \text{ResState}' = [\text{ResState} \text{ EXCEPT } ![r].Locked = \text{TRUE},$ 
18 │    $![r].Owner = t]$ 
19 │                                                                    │
20 │ RVTasksReleasesResource(t, r)  $\triangleq$ 
21 │    $\wedge \text{ResState}[r].Owner = t$ 
22 │    $\wedge \text{ResState}' = [\text{ResState} \text{ EXCEPT } ![r].Owner = \text{NoTask},$ 
23 │    $![r].Locked = \text{FALSE}]$ 
24 │                                                                    │
25 │ RVTasksGoesToWL(t, r)  $\triangleq$ 
26 │    $\wedge t \neq \text{ResState}[r].Owner$ 
27 │    $\wedge t \notin \text{ResState}[r].WL$ 
28 │    $\wedge \text{ResState}' = [\text{ResState} \text{ EXCEPT } ![r].WL = @ \cup \{t\}]$ 
29 │                                                                    │
30 │ RVTasksGoesFromWLtoOwnership(t, r)  $\triangleq$ 
31 │    $\wedge \text{ResState}[r].Locked = \text{FALSE}$ 
32 │    $\wedge t \in \text{ResState}[r].WL$ 
33 │    $\wedge \text{ResState}' = [\text{ResState} \text{ EXCEPT } ![r].Locked = \text{TRUE},$ 
34 │    $![r].Owner = t,$ 
35 │    $![r].WL = @ \setminus \{t\}]$ 
36 │ ───────────────────────────────────────────────────────────────────┐
37 │ RVNext  $\triangleq$ 
38 │    $\exists t \in \text{Task} :$ 
39 │      $\exists r \in \text{Resource} :$ 
40 │        $\vee \text{RVTasksLocksResource}(t, r)$ 
41 │        $\vee \text{RVTasksReleasesResource}(t, r)$ 
42 │        $\vee \text{RVTasksGoesToWL}(t, r)$ 
43 │        $\vee \text{RVTasksGoesFromWLtoOwnership}(t, r)$ 
44 │                                                                    │
45 │ RVSpec  $\triangleq$  RVInit  $\wedge \Box[\text{RVNext}]_{\text{ResState}}$ 
46 │ ───────────────────────────────────────────────────────────────────┐
47 │ THEOREM RVSpec  $\Rightarrow \Box \text{RVTypeOk}$ 
48 │ ───────────────────────────────────────────────────────────────────┐

```

Figure 3.3: The *ResourceView* module.

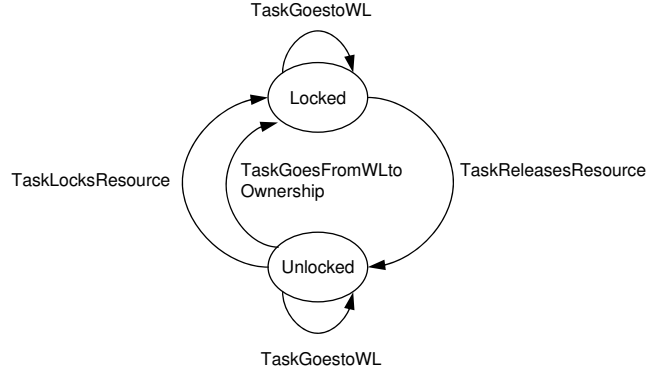


Figure 3.4: Resource lock transition diagram.

Variable *TaskOw* replicates the information related to the resources' allocation. It will be convenient for the write of expressions in further modules and could be used in an implementation.

There are no “new” actions, i.e., the actions defined result from merging the actions from modules *TaskView* and *ResourceView*. The module contains six actions:

- (i) *IRAScheduling*, from *TVScheduling*.
- (ii) *IRATaskLocksResource*, from *RVTaskLocksResource*.
- (iii) *IRATaskReleasesResource*, from *RVTaskReleasesResource*.
- (iv) *IRATaskGoesToWL*, a combination of actions *RVTaskGoesToWL* and *TVWait*.
- (v) *IRATaskGoesFromWLtoOwnership*, a combination of *TVSignal* and *RVTaskGoesFromWLtoOwnership*.
- (vi) *IRAPreemption*, from *TVPreemption*.

Note that the actions (i) and (vi), designated to be “from” an action of module *TaskView*, leave *ResState* (which is not specified in that module) unchanged. Equivalently, actions (ii) and (iii), “from” the *ResourceView* module, leave *TaskState* unchanged. Actions (iv) and (v) change both variables.

The less immediate step is the exact definition of the guards of actions (ii), (iii), (iv), and (v). Tuning them to correctly “synchronize” modules *TaskView* and *ResourceView* is an exercise where TLC can be of great help.

It is also appropriate to check that the guards defined for each action constitute the minimum set required. For example, action *IRATaskGoesFromWLtoOwnership* could contain the guard $TaskState[ut] = \text{“Waiting”}$, but that is not necessary. This is because the guard $t \in ResState[r].WL$ already implies it.

The invariant *IRABlocked*, defined at the bottom of the module, asserts that fact.

Figure 3.5, below, schematically illustrates the “updated” version of the diagram shown in Figure 3.1, on page 26. It manifestly hides many of the details of the specification. It becomes evident that diagrams and pictures are not a good way to specify systems. They are either too simple, or easily get too confusing, leaving people lost in the amalgam of boxes, pictures, and circles. Schematic drawings can (/should) be used as a complement to the specification, providing an initial simple illustration. Full description, however, should only be made with a precise language.

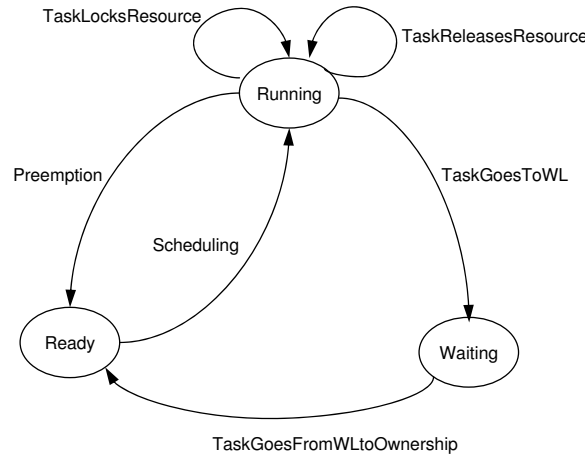


Figure 3.5: “Updated” task state transition diagram.

The liveness condition *IRALive* specifies weak fairness of action *IRAScheduling* and strong fairness of actions *IRATaskReleasesResource* and *IRATaskGoesFromWLtoOwnership*.

Weak fairness of an action *A* asserts that if *A* becomes for ever enabled, then an *A* step must eventually occur. Strong fairness of an action *B* asserts that if *B* is infinitely often enabled, then infinitely many *B* steps occur. Weak and strong fairness of an action *C* are equivalent iff *C* infinitely often disabled implies that either *C* eventually becomes forever disabled, or else infinitely many *C* steps occur. When strong and weak fairness are equivalent, the fairness property should be written as weak fairness.

In our specification, weak and strong fairness of actions *IRATaskReleasesResource* and *IRATaskGoesFromWLtoOwnership* are not equivalent. Intuitively, action *IRATaskReleasesResource* can be infinitely often disabled because steps *IRAPreemption* and *IRAScheduling* repeatedly occur. Action *IRATaskGoesFromWLtoOwnership* can be infinitely often disabled because steps *IRATaskLocksResource* occur.

The final part of module *InternalResourceAccess* contains the definition of some safety and liveness requirements. The invariant *IRALock* is defined as the conjunction of the four safety properties defined. The first, *IRASingleOwner*, asserts that the intersection of the *TaskOw[t].Owning* field of any two tasks *ti* and *tj* is empty, unless *ti = tj*. From the definition of variable *ResState*, it is obvious that a resource can only have one owner at a time, but this information could be wrongly replicated in variable *TaskOw*. Combining *IRASingleOwner* and *IRAOwnCorrect* asserts that the replication is correct. On the other side, *IRAWaitsRCorrect* asserts about the replication of field *TaskOw[t].WaitsR*. Invariant *IRABlocked* asserts that any task belonging to some resource waiting list is in the “waiting state”.

Specifications should contain liveness requirements. We defined property *IRARunTask*, specifying that every task should be infinitely often in the “running state”. The complete module is shown bellow.

```

1  ┌────────────────── MODULE InternalResourceAccess ───────────────────┐
2  │ CONSTANTS Task, Resource, NoTask, NoRes                             │
3  │                                                                 │
4  │ VARIABLES ResState, TaskState, TaskOw                             │
5  │ IRAvars  $\triangleq$   $\langle ResState, TaskState, TaskOw \rangle$            │
6  │                                                                 │
7  │ IRATypeOk  $\triangleq$                                              │
8  │    $\wedge ResState \in [Resource \rightarrow [Locked : BOOLEAN,$  │
9  │                                      $WL : SUBSET (Task),$  │
10 │                                      $Owner : Task \cup \{NoTask\}]]$  │
11 │                                                                 │
12 │    $\wedge TaskState \in [Task \rightarrow \{“Ready”, “Running”, “Waiting”\}]$  │
13 │                                                                 │
14 │    $\wedge TaskOw \in [Task \rightarrow [Owning : SUBSET (Resource),$  │
15 │                                      $WaitsR : Resource \cup \{NoRes\}]]$  │
16 │                                                                 │
17 │ IRAINit  $\triangleq$                                              │
18 │    $\wedge ResState = [r \in Resource \mapsto [Locked \mapsto FALSE,$  │
19 │                                      $WL \mapsto \{\},$  │
20 │                                      $Owner \mapsto NoTask]]$  │
21 │    $\wedge TaskState = [t \in Task \mapsto “Ready”]$  │
22 │    $\wedge TaskOw = [t \in Task \mapsto [Owning \mapsto \{\},$  │
23 │                                      $WaitsR \mapsto NoRes]]$  │
24 │ ───────────────────┐
25 │ IRAScheduling(t)  $\triangleq$  │
26 │   Notice that there are no extra conditions. Any number of tasks may be running. │
27 │    $\wedge TaskState[t] = “Ready”$  │
28 │    $\wedge TaskState' = [TaskState \text{ EXCEPT } ![t] = “Running”]$  │
29 │    $\wedge \text{UNCHANGED } \langle ResState, TaskOw \rangle$  │
30 │                                                                 │
31 │ IRATaskLocksResource(t, r)  $\triangleq$  │
32 │    $\wedge TaskState[t] = “Running”$  │
33 │    $\wedge ResState[r].Locked = FALSE$  │
34 │    $\wedge ResState' = [ResState \text{ EXCEPT } ![r].Locked = TRUE,$  │
35 │                                      $![r].Owner = t]$  │
36 │    $\wedge TaskOw' = [TaskOw \text{ EXCEPT } ![t].Owning = @ \cup \{r\}]$  │
37 │    $\wedge \text{UNCHANGED } \langle TaskState \rangle$ 

```

```

39  $IRATaskReleasesResource(t, r) \triangleq$ 
40    $\wedge ResState[r].Owner = t$ 
41    $\wedge TaskState[t] = \text{"Running"}$ 
42    $\wedge ResState' = [ResState \text{ EXCEPT } ![r].Owner = NoTask,$ 
43      $![r].Locked = FALSE]$ 
44    $\wedge TaskOw' = [TaskOw \text{ EXCEPT } ![t].Owning = @ \setminus \{r\}]$ 
45    $\wedge \text{UNCHANGED } \langle TaskState \rangle$ 

47  $IRATaskGoesToWL(t, r) \triangleq$ 
48    $\wedge TaskState[t] = \text{"Running"}$ 
49    $\wedge t \neq ResState[r].Owner$ 
50    $\wedge ResState' = [ResState \text{ EXCEPT } ![r].WL = @ \cup \{t\}]$ 
51    $\wedge TaskState' = [TaskState \text{ EXCEPT } ![t] = \text{"Waiting"}]$ 
52    $\wedge TaskOw' = [TaskOw \text{ EXCEPT } ![t].WaitsR = r]$ 

54  $IRATaskGoesFromWLtoOwnership(t, r) \triangleq$ 
55    $\wedge ResState[r].Locked = FALSE$ 
56    $\wedge t \in ResState[r].WL$ 
57    $\wedge ResState' = [ResState \text{ EXCEPT } ![r].Locked = TRUE,$ 
58      $![r].Owner = t,$ 
59      $![r].WL = @ \setminus \{t\}]$ 
60    $\wedge TaskState' = [TaskState \text{ EXCEPT } ![t] = \text{"Ready"}]$ 
61    $\wedge TaskOw' = [TaskOw \text{ EXCEPT } ![t].WaitsR = NoRes,$ 
62      $![t].Owning = @ \cup \{r\}]$ 

64  $IRAPreemption(t) \triangleq$ 
65    $\wedge TaskState[t] = \text{"Running"}$ 
66    $\wedge TaskState' = [TaskState \text{ EXCEPT } ![t] = \text{"Ready"}]$ 
67    $\wedge \text{UNCHANGED } \langle ResState, TaskOw \rangle$ 
68 |-----|

69  $IRANext \triangleq$ 
70    $\exists t \in Task :$ 
71      $\vee \exists r \in Resource :$ 
72        $\vee IRATaskLocksResource(t, r)$ 
73        $\vee IRATaskReleasesResource(t, r)$ 
74        $\vee IRATaskGoesToWL(t, r)$ 
75        $\vee IRATaskGoesFromWLtoOwnership(t, r)$ 
76      $\vee IRAScheduling(t)$ 
77      $\vee IRAPreemption(t)$ 

79  $IRALive \triangleq$ 
80    $\forall t \in Task :$ 
81      $\wedge \forall r \in Resource :$ 
82        $\wedge SF_{IRAvars}(IRATaskReleasesResource(t, r))$ 
83        $\wedge SF_{IRAvars}(IRATaskGoesFromWLtoOwnership(t, r))$ 
84      $\wedge WF_{IRAvars}(IRAScheduling(t))$ 

86  $IRASpec \triangleq IRANit \wedge \Box[IRANext]_{IRAvars}$ 

88  $IRALiveSpec \triangleq IRASpec \wedge IRALive$ 
89 |-----|

90  $IRASingleOwner \triangleq \forall ti, tj \in Task :$ 
91    $TaskOw[ti].Owning \cap TaskOw[tj].Owning \neq \{\} \Rightarrow ti = tj$ 

93  $IRABlocked \triangleq \forall t \in Task :$ 
94    $(\exists r \in Resource : t \in ResState[r].WL) \equiv (TaskState[t] = \text{"Waiting"})$ 

96  $IRAWaitsRCorrect \triangleq \forall t \in Task :$ 
97    $TaskOw[t].WaitsR \neq NoRes \Rightarrow t \in ResState[TaskOw[t].WaitsR].WL$ 

```

```

99   $IRAOwnCorrect \triangleq \forall r \in Resource :$ 
100     $ResState[r].Locked = TRUE \Rightarrow r \in TaskOw[ResState[r].Owner].Owning$ 
102   $IRALock \triangleq IRASingleOwner \wedge IRABlocked \wedge IRAWaitsRCorrect \wedge IRAOwnCorrect$ 
104   $IRARunTask \triangleq \forall t \in Task : \Box \Diamond (TaskState[t] = \text{"Running"})$ 
105  └──────────────────────────────────────────────────────────────────────────────────┘

```

Figure 3.6: The *InternalResourceAccess* module.

3.3.1 Avoiding Circular Wait Conditions

Module *InternalResourceAccess* asserts no theorems about the specification. This is because Model checking it with TLC it is found a problem: the system might deadlock. In fact, the specification allows behaviors that might lead to situations where every task waits on a resource that is in turn owned by another task. Consequently, no task can proceed its execution and the system deadlocks. An example of such situation is illustrated in Figure 3.7.

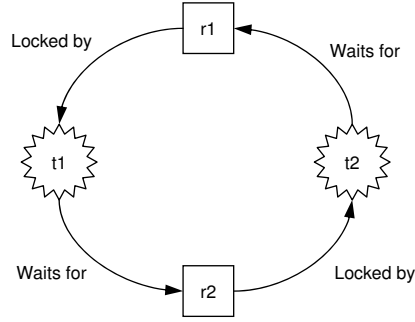


Figure 3.7: Circular wait condition.

How can the error be corrected? First of all, it all depends of what is intended. Namely, whether it is considered that the system is responsible for avoiding such situation or not. Intuitively, the specification of module *InternalResourceAccess* describes the interaction between resources and an environment of tasks trying to access them.

In a real implementation, it is highly important to specify which problems are to be dealt by the system and which are dealt by the environment. A specification is called a complete-system specification when it is satisfied by a behavior that represents the correct operation of both the system and its environment. On the contrary, an open-system specification can serve as a contract between a user and an implementer. For the definition of open-system specifications TLA⁺ provides the temporal operator $\vdash$. If we write a specification as the

composition of an environment specification E and a system specification M , we may write $E \triangleleft M$. Here, $E \triangleleft M$ asserts that M remains true at least one step longer than E does, remaining true forever if E does.

Writing the specification of module *InternalResourceAccess* as an open-system specification would force us to rearrange it. We will leave the module unaltered and solve the system deadlock problem by disabling any steps that might lead to *circular wait conditions*.

By circular wait conditions we mean situations where any subset of tasks ST (with at least two tasks) can form a closed loop where every task in ST waits on a resource that is in turn owned by another task of ST . Such a loop can only be closed by an *IRATaskGoesFromWLtoOwnership*(t, r) step. Hence, *IRATaskGoesFromWLtoOwnership*(t, r) is the action to constrain. We will add to its definition the conjunct:

$$\neg(\text{ResState}[r].\text{Locked} = \text{TRUE} \wedge (\text{IsCyclicIn}(\langle t, r \rangle, \text{GraphLW}(\langle t, r \rangle)))) \quad (3.1)$$

The definition of *IsCyclicIn* and *GraphLW* is made in module *ResAllocGraph*, shown in Figure 3.8. It defines a graph whose nodes represent relations between tasks and resources. There are two kinds of nodes. Every ordered pair of resource r and task t , $\langle r, t \rangle$, such that r is locked by t is a node *NodeL*. Every ordered pair of task t and resource r , $\langle t, r \rangle$, such that t waits for r is a node *NodeW*. Because the goal is to avoid possible circular wait conditions caused by an *IRATaskGoesFromWLtoOwnership*(t, r) step, the pair *fut* of the task t and resource r in question is also a node of the graph. The union of *NodeL*, *NodeW*, and *fut* is *NodeLW*(*fut*). Set *EdgeLW* is the set of edges of the graph. An edge is an ordered pair of nodes $\langle ni, nj \rangle$, such that $ni[2]$ equals $nj[1]$. It represents a connection between two nodes of different kind where they have either a task t or a resource r in common. To detect closed loops, it is checked whether the insertion of node *fut* creates a path connecting several nodes such that it is possible to “return” to the initial node *fut*. This is represented by operator *IsCyclicIn*(n, G), where n is replaced by node *fut* and G is the described graph. Figure 3.9 depicts the situation.

The LOCAL INSTANCE statements are used to incorporate into the module the definitions of modules *Naturals*, *Sequences*, and *FiniteSets*. The modifier LOCAL makes the definitions following it local to the module. Like that, a module that extends or instantiates module *ResAllocGraph* does not obtain the definitions of the *Naturals*, *Sequences*, and *FiniteSets* modules.

3.3.2 The Resource Access Specification

In order to leave module *InternalResourceAccess* unaltered, it is created module *MCInternalResourceAccess* that extends module *InternalResourceAccess* and defines action *MCIRATaskGoesToWL* as a conjunction of action *IRATaskGoesToWL* and (3.1). A new next-state relation *MCIRANext* is defined. It is similar to *IRANext*, except that action *MCIRATaskGoesToWL* replaces *IRATaskGoesToWL*. The module is shown in Figure 3.10. The theorems contained in it and the definition of *Perms* will be explained in Section 3.5.3.

```

1  ┌────────────────────────── MODULE ResAllocGraph ───────────────────────────┐
2  LOCAL INSTANCE Naturals
3  LOCAL INSTANCE Sequences
4  LOCAL INSTANCE FiniteSets

6  CONSTANTS Task, Resource, NoTask, NoRes

8  VARIABLES ResState, TaskOw
9  ┌────────────────────────────────────────────────────────────────────────────────┐
10 NodeL  $\triangleq$   $\{n \in \text{Resource} \times \text{Task} :$ 
11                $\wedge \text{ResState}[n[1]].\text{Locked} = \text{TRUE}$ 
12                $\wedge n[2] = \text{ResState}[n[1]].\text{Owner}\}$ 

14 NodeW  $\triangleq$   $\{n \in \text{Task} \times \text{Resource} : \text{TaskOw}[n[1]].\text{WaitsR} = n[2]\}$ 

16 NodeLW(fut)  $\triangleq$  NodeL  $\cup$  NodeW  $\cup$   $\{\text{fut}\}$ 

18 EdgeLW(N)  $\triangleq$   $\{e \in N \times N : e[1][2] = e[2][1]\}$ 

20 GraphLW(fut)  $\triangleq$  [node  $\mapsto$  NodeLW(fut),
21                   edge  $\mapsto$  EdgeLW(NodeLW(fut))]

23 MySeq(S)  $\triangleq$  UNION  $\{[1 \dots n \rightarrow S] : n \in 1 \dots (\text{Cardinality}(S) + 1)\}$ 

25 MyPath(G)  $\triangleq$   $\{p \in \text{MySeq}(G.\text{node}) :$ 
26                $\wedge p \neq \langle \rangle$ 
27                $\wedge \forall i \in 1 \dots (\text{Len}(p) - 1) : \langle p[i], p[i + 1] \rangle \in G.\text{edge}\}$ 

29 IsCyclicIn(n, G)  $\triangleq$   $\exists p \in \text{MyPath}(G) :$ 
30                $((p[1] = n) \wedge (p[\text{Len}(p)] = n) \wedge (\text{Len}(p) > 1) \wedge (p[2] \neq n))$ 

32 IsConnectedTo(t, n, G)  $\triangleq$ 
33                $\exists p \in \text{MyPath}(G) : (p[1] = n \wedge p[\text{Len}(p)][2] = t)$ 
34 └────────────────────────────────────────────────────────────────────────────────┘

```

Figure 3.8: The *ResAllocGraph* module.

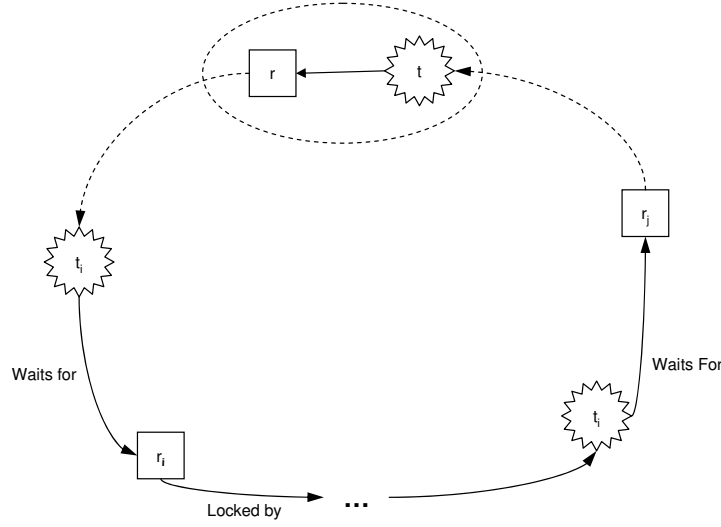


Figure 3.9: Avoiding circular wait conditions.

3.4 The Resource Management Mechanism

A resource access management mechanism that implements the specification described in module *InternalResourceAccess* is now specified.

The system behavior is depicted in Figure 3.11. Each task t can either request a resource r_i , or release a resource r_j that already owns. The system processes these requests as specified by actions *SRMTaskRequestsResource* and *SRMTaskReleasesResource*, respectively.

When a task t releases a resource r , the value of $ResNotify[r]$ is set to TRUE. Variable $ResNotify$ is defined as a function that maps every resource to a boolean value, $ResNotify \in [Resource \rightarrow BOOLEAN]$.

The value of $ResNotify[r]$ is reset to FALSE with a *SRMNotifyResourceUnlock* step. Here, if there are no tasks in the waiting list of resource r nothing else is changed. Otherwise, a task $woken$ from r 's waiting list is removed from there, granted the ownership of r , and made ready.

The definition of action *SRMTaskRequestsResource* contains two different branches. The first represents the lock of a resource r by a task t . If locking resource r is not possible, task t is blocked and placed in r 's waiting list. This is represented in the second branch. Note, however, that the two branches are not mutually exclusive.² When (a) resource r is not locked, (b) $ResNotify[r]$ is FALSE, and (c) r 's waiting list is not empty, both possibilities are enabled. Figure 3.12, below, helps to illustrate the conditions defined.

In comparison with module *InternalResourceAccess*, it can be said that

²This is an example of a situation where it should not be used the CASE construct, as mentioned in Section 2.2.3 (page 21).

```

3  | MODULE MCInternalResourceAccess
4  | EXTENDS InternalResourceAccess, TLC
5
6  | LOCAL INSTANCE ResAllocGraph
7  | LOCAL INSTANCE TaskView
8  | LOCAL INSTANCE ResourceView
9
10 | LOCAL Perms  $\triangleq$  Permutations(Task)  $\cup$  Permutations(Resource)
11
12 | MCIRATaskGoesToWL(t, r)  $\triangleq$ 
13 |    $\wedge$  IRATaskGoesToWL(t, r)
14 |   “No circular wait condition”
15 |    $\wedge \neg(\text{ResState}[r].\text{Locked} = \text{TRUE} \wedge (\text{IsCyclicIn}(\langle t, r \rangle, \text{GraphLW}(\langle t, r \rangle))))$ 
16 |
17 | MCIRANext  $\triangleq$ 
18 |    $\exists t \in \text{Task} :$ 
19 |      $\vee \exists r \in \text{Resource} :$ 
20 |        $\vee \text{IRATaskLocksResource}(t, r)$ 
21 |        $\vee \text{IRATaskReleasesResource}(t, r)$ 
22 |        $\vee \text{MCIRATaskGoesToWL}(t, r)$ 
23 |        $\vee \text{IRATaskGoesFromWLtoOwnership}(t, r)$ 
24 |        $\vee \text{IRAScheduling}(t)$ 
25 |        $\vee \text{IRAPreemption}(t)$ 
26 |
27 | MCIRASpec  $\triangleq$  IRAINit  $\wedge \Box[\text{MCIRANext}]_{\text{IRAvars}}$ 
28 |
29 | MCIRALiveSpec  $\triangleq$  MCIRASpec  $\wedge$  IRALive
30 |
31 | THEOREM MCIRASpec  $\Rightarrow \Box(\text{IRATypeOk} \wedge \text{IRALock})$ 
32 | THEOREM MCIRASpec  $\Rightarrow (\text{IRASpec} \wedge \text{RVSpec} \wedge \text{TVSpec})$ 
33 |
34 | THEOREM MCIRALiveSpec  $\Rightarrow \text{IRARunTask}$ 

```

Figure 3.10: The *MCInternalResourceAccess* module.

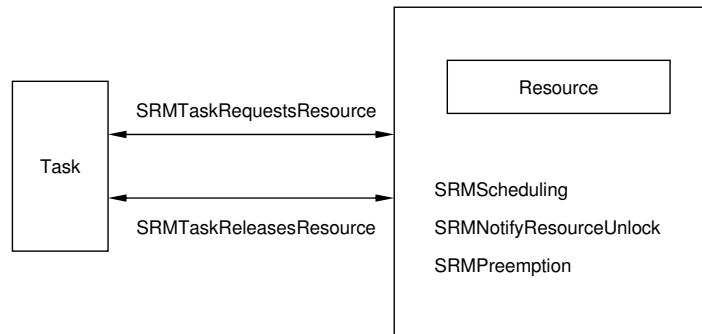


Figure 3.11: Resource management representation.

Res. Locked	Res. W.L.	Res. Notify	"Task Locks"	"Task goes to WL"
TRUE	Empty	TRUE	✗	✓
		FALSE	✗	✓
	Not empty	TRUE	✗	✓
		FALSE	✗	✓
FALSE	Empty	TRUE	✗	✓
		FALSE	✓	✗
	Not empty	TRUE	✗	✓
		FALSE	✓	✓

Figure 3.12: Guarding conditions for the two branches of action *SRMTaskRequestsResource*.

action *SRMTaskRequestsResource* incorporates actions *IRATaskLocksResource* and *IRATaskGoesToWL*, and that action *SRMNotifyResourceUnlock* replaces action *IRATaskGoesFromWLtoOwnership*. In Section 3.5.3, on page 43, it is shown how to check that a specification implements another.

The specification appears in module *SimpleResourceManagement*, in Figure 3.13 bellow. It also contains the definition of the liveness condition *SRMLive*, invariant *SRMNotification*, and property *SRMRunTask*. The definition of *SRMLive* is analogous to the definition of *IRALive* (which has been explained on page 31). The invariant *SRMNotification* asserts that a resource *r* cannot be locked while *ResNotify[r]* is true. The temporal property *SRMRunTask* is equivalent to *IRARunTask*, from module *InternalResourceAccess*.

```

1  ┌────────────────── MODULE SimpleResourceManagement ───────────────────┐
2  │ CONSTANTS Task, Resource, NoTask, NoRes                                │
3  │                                                                           │
4  │ VARIABLES ResState, TaskState, TaskOw, ResNotify                      │
5  │ SRMvars  $\triangleq$   $\langle$  ResState, TaskState, TaskOw, ResNotify  $\rangle$           │
6  │                                                                           │
7  │ SRMTypeOk  $\triangleq$                                                        │
8  │    $\wedge$  ResState  $\in$  [Resource  $\rightarrow$  [Locked : BOOLEAN ,           │
9  │                                           WL : SUBSET (Task),           │
10 │                                           Owner : Task  $\cup$  {NoTask}]]          │
11 │                                                                           │
12 │    $\wedge$  TaskState  $\in$  [Task  $\rightarrow$  { "Ready", "Running", "Waiting" }] │
13 │                                                                           │
14 │    $\wedge$  TaskOw  $\in$  [Task  $\rightarrow$  [Owning : SUBSET (Resource),           │
15 │                                           WaitsR : Resource  $\cup$  {NoRes}]]          │
16 │                                                                           │
17 │    $\wedge$  ResNotify  $\in$  [Resource  $\rightarrow$  BOOLEAN ]                      │
18 │                                                                           │
19 │ SRMInit  $\triangleq$                                                          │
20 │    $\wedge$  ResState = [r  $\in$  Resource  $\mapsto$  [Locked  $\mapsto$  FALSE,           │
21 │                                           WL  $\mapsto$  {},           │
22 │                                           Owner  $\mapsto$  NoTask]]          │
23 │    $\wedge$  TaskState = [t  $\in$  Task  $\mapsto$  "Ready"]                          │
24 │    $\wedge$  TaskOw = [t  $\in$  Task  $\mapsto$  [Owning  $\mapsto$  {}],                  │

```

```

25                               WaitsR  $\mapsto$  NoRes]]
26    $\wedge$  ResNotify =  $[r \in \text{Resource} \mapsto \text{FALSE}]$ 
27 |-----|
28   SRMScheduling( $t$ )  $\triangleq$ 
29   Note that any number of tasks may be running.
30    $\wedge$  TaskState[ $t$ ] = "Ready"
31    $\wedge$  TaskState' = [TaskState EXCEPT ![ $t$ ] = "Running"]
32    $\wedge$  UNCHANGED  $\langle$ ResState, TaskOw, ResNotify $\rangle$ 

34   SRMTaskRequestsResource( $t, r$ )  $\triangleq$ 
35   Note that the two branches are not mutually exclusive.
36    $\wedge$  TaskState[ $t$ ] = "Running"
37    $\wedge$  Task locks resource
38    $\vee$  ( $\text{ResState}[r].\text{Locked} = \text{FALSE} \wedge \text{ResNotify}[r] = \text{FALSE}$ )
39    $\wedge$  ResState' = [ResState EXCEPT ![ $r$ ].Locked = TRUE,
40    $\quad$  ![ $r$ ].Owner =  $t$ ]
41    $\wedge$  TaskOw' = [TaskOw EXCEPT ![ $t$ ].Owning =  $@ \cup \{r\}$ ]
42    $\wedge$  UNCHANGED  $\langle$ TaskState, ResNotify $\rangle$ 
43   Task goes to WL
44    $\vee \neg(\text{ResState}[r].\text{Locked} = \text{FALSE} \wedge \text{ResState}[r].\text{WL} = \{\} \wedge \text{ResNotify}[r] = \text{FALSE})$ 
45    $\wedge t \neq \text{ResState}[r].\text{Owner}$ 
46    $\wedge$  ResState' = [ResState EXCEPT ![ $r$ ].WL =  $@ \cup \{t\}$ ]
47    $\wedge$  TaskState' = [TaskState EXCEPT ![ $t$ ] = "Waiting"]
48    $\wedge$  TaskOw' = [TaskOw EXCEPT ![ $t$ ].WaitsR =  $r$ ]
49    $\wedge$  UNCHANGED  $\langle$ ResNotify $\rangle$ 

51   SRMTaskReleasesResource( $t, r$ )  $\triangleq$ 
52    $\wedge$  ResState[ $r$ ].Owner =  $t$ 
53    $\wedge$  TaskState[ $t$ ] = "Running"
54    $\wedge$  ResState' = [ResState EXCEPT ![ $r$ ].Owner = NoTask,
55    $\quad$  ![ $r$ ].Locked = FALSE]
56    $\wedge$  TaskOw' = [TaskOw EXCEPT ![ $t$ ].Owning =  $@ \setminus \{r\}$ ]
57    $\wedge$  ResNotify' = [ResNotify EXCEPT ![ $r$ ] = TRUE]
58    $\wedge$  UNCHANGED  $\langle$ TaskState $\rangle$ 

60   SRMNotifyResourceUnlock( $wokent, r$ )  $\triangleq$ 
61    $\wedge$  ResNotify[ $r$ ] = TRUE
62    $\wedge$  IF ResState[ $r$ ].WL  $\neq \{\}$ 
63   THEN
64   If there are tasks waiting one of the waiting tasks is granted the ownership of the
65   resource.
66    $\wedge$  wokent  $\in$  ResState[ $r$ ].WL
67    $\wedge$  ResState' = [ResState EXCEPT ![ $r$ ].Locked = TRUE,
68    $\quad$  ![ $r$ ].Owner = wokent,
69    $\quad$  ![ $r$ ].WL =  $@ \setminus \{wokent\}$ ]
70    $\wedge$  TaskState' = [TaskState EXCEPT ![wokent] = "Ready"]
71    $\wedge$  TaskOw' = [TaskOw EXCEPT ![wokent].WaitsR = NoRes,
72    $\quad$  ![wokent].Owning =  $@ \cup \{r\}$ ]
73   ELSE
74   Otherwise simply "consume" the notify signal.
75   UNCHANGED  $\langle$ ResState, TaskState, TaskOw $\rangle$ 
76    $\wedge$  ResNotify' = [ResNotify EXCEPT ![ $r$ ] = FALSE]

78   SRMPreemption( $t$ )  $\triangleq$ 
79    $\wedge$  TaskState[ $t$ ] = "Running"
80    $\wedge$  TaskState' = [TaskState EXCEPT ![ $t$ ] = "Ready"]
81    $\wedge$  UNCHANGED  $\langle$ ResState, TaskOw, ResNotify $\rangle$ 

```

82	
83	$SRMNext \triangleq$
84	$\exists t \in Task :$
85	$\vee \exists r \in Resource :$
86	$\vee SRMTaskRequestsResource(t, r)$
87	$\vee SRMTaskReleasesResource(t, r)$
88	$\vee SRMNotifyResourceUnlock(t, r)$
89	$\vee SRMScheduling(t)$
90	$\vee SRMPreemption(t)$
92	$SRMLive \triangleq$
93	$\forall t \in Task :$
94	$\wedge \forall r \in Resource :$
95	$\wedge SF_{SRMvars}(SRMTaskReleasesResource(t, r))$
96	$\wedge SF_{SRMvars}(SRMNotifyResourceUnlock(t, r))$
97	$\wedge WF_{SRMvars}(SRMScheduling(t))$
99	$SRMSpec \triangleq SRMInit \wedge \square[SRMNext]_{SRMvars}$
101	$SRMLiveSpec \triangleq SRMSpec \wedge SRMLive$
102	
103	$SRMNotification \triangleq \forall r \in Resource :$
104	$ResNotify[r] = TRUE \Rightarrow ResState[r].Locked = FALSE$
106	$SRMRunTask \triangleq \forall t \in Task : \square\Diamond(TaskState[t] = "Running")$
107	

Figure 3.13: The *SimpleResourceManagement* module.

3.4.1 Exclusion of Circular Wait Conditions

The specification in module *SimpleResourceManagement* allows behaviors that lead to deadlock, similarly to the situation described in Section 3.3.1 for module *InternalResourceAccess*. An analogous procedure is followed to solve the problem.

Figure 3.14 shows module *MCSimpleResourceManagement* created. It extends module *SimpleResourceManagement* and defines action *MCSRMTasksRequestsResource* as a conjunction of action *SRMTaskRequestsResource* and equation (3.1) on page 35.

A new next-state relation *MCSRNext* is defined. It is similar to *SRMNext*, except that action *SRMTaskRequestsResource* is replaced by action *MCSRMTasksRequestsResource*. The theorems contained in the module are discussed next, in Section 3.5.3.

3.5 Safety and Liveness Checking

All the models contained in this chapter were model checked with TLC. This section summarizes such process.

```

1  ┌────────────────── MODULE MCSimpleResourceManagement ───────────────────┐
2  EXTENDS SimpleResourceManagement, TLC

4  LOCAL INSTANCE ResAlocGraph
5  LOCAL INSTANCE TaskView
6  LOCAL INSTANCE ResourceView
7  LOCAL INSTANCE InternalResourceAccess

9  LOCAL Perms  $\triangleq$  Permutations(Task)  $\cup$  Permutations(Resource)

11 MCSRMTasksRequestsResource(t, r)  $\triangleq$ 
12    $\wedge$  SRMTasksRequestsResource(t, r)
13   “No circular wait condition”
14    $\wedge \neg(\text{ResState}[r].\text{Locked} = \text{TRUE} \wedge (\text{IsCyclicIn}(\langle t, r \rangle, \text{GraphLW}(\langle t, r \rangle))))$ 
15 ───────────────────────────────────────────────────────────────────────────┐
16 MCSRMMNext  $\triangleq$ 
17    $\exists t \in \text{Task} :$ 
18      $\vee \exists r \in \text{Resource} :$ 
19        $\vee \text{MCSRMTasksRequestsResource}(t, r)$ 
20        $\vee \text{SRMTasksReleasesResource}(t, r)$ 
21        $\vee \text{SRMNotifyResourceUnlock}(t, r)$ 
22        $\vee \text{SRMScheduling}(t)$ 
23        $\vee \text{SRMPreemption}(t)$ 

25 MCSRMSpec  $\triangleq$  SRMInit  $\wedge \Box[\text{MCSRMMNext}]_{\text{SRMvars}}$ 

27 MCSRMLiveSpec  $\triangleq$  MCSRMSpec  $\wedge$  SRMLive
28 ───────────────────────────────────────────────────────────────────────────┐
29 THEOREM MCSRMSpec  $\Rightarrow \Box(\text{SRMTypeOk} \wedge \text{SRMNotification})$ 
30 THEOREM MCSRMSpec  $\Rightarrow (\text{SRMSpec} \wedge \text{IRASpec} \wedge \text{RVSpec} \wedge \text{TVSpec})$ 

32 THEOREM MCSRMLiveSpec  $\Rightarrow \text{SRMRunTask}$ 
33 ───────────────────────────────────────────────────────────────────────────┐

```

Figure 3.14: The *MCSimpleResourceManagement* module.

3.5.1 The Initial Models

Modules *TaskView* and *ResourceView* are quite simple. The assertions made are that they imply, respectively, invariants *TVTypeOk* and *RVTypeOk*. These invariants constitute safety properties. TLC found no errors in the specifications.

As mentioned, model checking modules *SimpleResourceManagement* and *InternalResourceAccess*, TLC finds behaviors that lead to deadlock. For that reason, no theorems are asserted in these models. They are checked in their respective “MCversion”. This is described in 3.5.3, but, before proceeding, it is convenient to introduce the concept of *machine closure*.

3.5.2 Machine Closure and TLC’s Liveness Checking

A TLA^+ specification *Spec* has the general form

$$\text{Spec} \triangleq \text{Init} \wedge \Box[\text{Next}]_{\text{vars}} \wedge \text{Liveness} \quad (3.2)$$

Here, *Init* is expected to constrain the initial state, *Next* to constrain what steps may occur, and *Liveness* to describe what must eventually happen.

A specification of the form (3.2) is called *machine closed* iff *Liveness* constraints neither the initial state nor what steps may occur.

Specification (3.2) is guaranteed to be machine closed if *Liveness* is the conjunction of weak and/or strong fairness properties for subactions of *Next*. An action *A* is a subaction of *Next* iff every *A* step is a *Next* step.

When TLC models checks a specification (3.2) that contains property *Prop* of the form

$$\text{Prop} \triangleq \text{PInit} \wedge \Box[\text{PNext}]_{\text{vars}} \wedge \text{PLiveness} \quad (3.3)$$

TLC does not actually verify that the specification implies the property, i.e., that $\text{Spec} \Rightarrow \text{Prop}$. Instead, it checks that (i) the safety part of the specification implies the safety part of the property, and (ii) the specification implies the liveness part of the property, i.e.,

$$(i) \text{Init} \wedge \Box[\text{Next}]_{\text{vars}} \Rightarrow \text{PInit} \wedge \Box[\text{PNext}]_{\text{vars}}$$

$$(ii) \text{Spec} \Rightarrow \text{PLiveness}$$

This means that TLC cannot be used to check that a non-machine-closed specification satisfies a safety property.

3.5.3 Proving Implementation

The explanation in Section 3.5.2 should make clear the form of the theorems expressed at the bottom of modules *MCSimpleResourceManagement* and *MCIInternalResourceAccess*.

For example, in module *MCSimpleResourceManagement* it were specified the theorems:

THEOREM $MCSRMSpec \Rightarrow \Box(SRMTypeOk \wedge SRMNotification)$

THEOREM $MCSRMLiveSpec \Rightarrow SRMRunTask$

The distinction in the left side of the implications is now clear. The first expression involves only safety properties, while the second concerns a liveness requirement.

Module *MCSimpleResourceManagement* contains yet another theorem, namely:

THEOREM $MCSRMSpec \Rightarrow (SRMSpec \wedge IRASpec \wedge RVSpec \wedge TVSpec)$

which asserts that specification *MCSRMSpec* implies all the other four specifications listed. This means that every behavior satisfying *MCSRMSpec* also satisfies *SRMSpec*, *IRASpec*, *RVSpec*, and *TVSpec*. This is the way to check that a system implements its specification.

In TLA^+ , implementation is implication. TLA^+ makes no distinction between system descriptions and specifications; they are both formulas. This uniformity distinguishes TLA^+ from other logics, languages, or other specification mechanisms. In TLA^+ , there is a single logic in which everything is specified.

The configuration file to model check such a module has the usual form. Figure 3.15 shows a possible configuration to model check module *MCSimpleResourceManagement*.

```
(* ..... *)
(*      CONFIG MCSimpleResourceManagement.cfg      *)
(* ..... *)

SPECIFICATION  MCSRMLiveSpec

CONSTANTS      Resource = {r1,r2}
                Task     = {t1,t2,t3,t4}
                NoTask   = NoTask
                NoRes    = NoRes

INVARIANT      SRMTypeOk  SRMNotification

PROPERTIES     SRMSpec  IRASpec  RVSpec  TVSpec  SRMRunTask

SYMMETRY       Perms

(* ..... *)
```

Figure 3.15: Configuration file for module *MCSimpleResourceManagement*.

In module *MCSimpleResourceManagement* the following statement can be found:

$$Perms \triangleq Permutations(Task) \cup Permutations(Resource)$$

SIMMETRY Perms

3.5.4 A Subtle Difference

$$\begin{array}{l}
DiffSRMNotifyResourceUnlock(r) \triangleq \\
\quad \wedge ResNotify[r] = TRUE \\
\quad \wedge IF ResState[r].WL \neq \{\} \\
\quad THEN \\
\quad \quad \exists woken \in ResState[r].WL : \\
\quad \quad \quad \wedge ResState' = [ResState \text{ EXCEPT } ![r].Locked = TRUE, \\
\quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad ![r].Owner = woken, \\
\quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad ![r].WL = @ \setminus \{woken\}] \\
\quad \quad \quad \wedge TaskState' = [TaskState \text{ EXCEPT } ![woken] = "Ready"] \\
\quad \quad \quad \wedge TaskOw' = [TaskOw \text{ EXCEPT } ![woken].WaitsR = NoRes, \\
\quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad ![woken].Owning = @ \cup \{r\}] \\
\quad ELSE \\
\quad \quad \wedge UNCHANGED \langle ResState, TaskState, TaskOw \rangle \\
\quad \wedge ResNotify' = [ResNotify \text{ EXCEPT } ![r] = FALSE]
\end{array}$$

Like that, the respective disjunct in the next-state relation would be:

And the conjunct in definition of *SRMLive* becomes:

With such a definition of the action, model checking module *MCSimpleResourceManagement* does not find a violation of any safety property. However, when checking the liveness property *SRMRunTask* TLC reports an error. This happens because with the definition shown in Figure 3.16, the expression of the liveness conjunct does not include the *Task* set. Therefore, TLC is able to find a behavior where a task stays blocked for ever. Like this example gives evidence, it is a good policy to move all the quantifiers to the outside of the definition of an action.

3.6 Summary

In this chapter, starting from two very simple modules, *TaskView* and *ResourceView*, we gradually built a mechanism for mutually exclusive access of tasks to resources. The module presented last, *MCSimpleResourceManagement*, implements the specifications of modules *TaskView*, *ResourceView*, *InternalResourceAccess*, and *SimpleResourceManagement*. It is our belief that it would have been much more difficult to develop such a specification directly from scratch, without starting with simpler systems. Figure 3.17 lists the modules presented throughout the chapter.

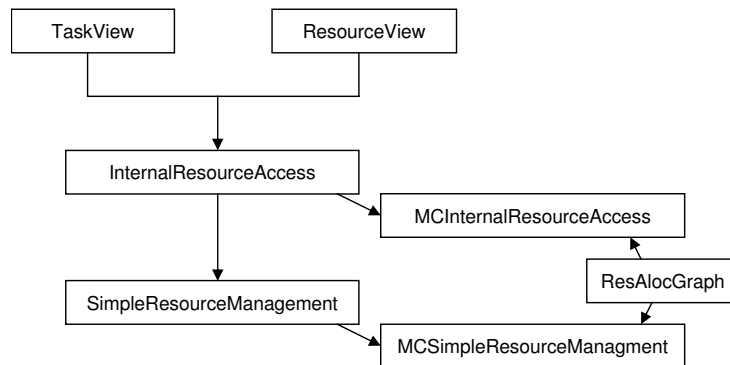


Figure 3.17: Modules presented in this chapter.

Chapter 4

Priority Handling

The correct behavior of a real-time application depends not only of the values produced by the computation, but also on the time at which they are produced [31]. Traditionally, to handle time constraints tasks are given priorities and scheduled according to some priority based protocol.

In this chapter, the resource access management mechanism developed in Chapter 3 is extended in order to support priority handling.

4.1 Adding Priorities to Tasks

A good way to develop a complex system is to start with a simple representation and gradually add detail and complexity to it. To start with, we will take the module *MCInternalResourceAccess*, shown in Figure 3.10 on page 38, and extend it, adding a new variable to the specification. This variable is called *TaskPri* and maps each task to a certain priority, defined as an integer from 1 (one) to *PriMax*.

The new specification constraints the behavior of the system in two situations. First, a task *t* can only lock a resource *r* if *r*'s waiting list is empty or if *t*'s priority is higher than the priority of any of the tasks in *r*'s waiting list. Action *PRATaskLocksResource* represents this. Second, a task *t* sitting in a resource's waiting list can only become its owner if *t*'s priority is higher than the priority of any of the tasks in the resource's waiting list. This is represented in action *PRATaskGoesFromWLtoOwnership*.

The remaining of the behavior specification stays unaltered. Module *PrioResourceAccess* shown next contains the specification. The definition of *PRARun*, at the end of the module, will be discussed in Section 4.1.1.

```
1  ┌────────────────── MODULE PrioResourceAccess ───────────────────┐
2  │ EXTENDS Naturals, FiniteSets, MCInternalResourceAccess          │
3  │                                                                     │
4  │ CONSTANTS PriMax                                                  │
```

6 VARIABLES $TaskPri$

8 $PRAvars \triangleq \langle ResState, TaskState, TaskOw, TaskPri \rangle$

10 $PRATypeOk \triangleq$
 11 $\quad \wedge IRATypeOk$
 12 $\quad \wedge TaskPri \in [Task \rightarrow 1 \dots PriMax]$

14 $PRAInit \triangleq$
 15 $\quad \wedge IRAInit$
 16 $\quad \wedge TaskPri \in [Task \rightarrow 1 \dots PriMax]$

18 $PRAScheduling(t) \triangleq$
 19 $\quad \wedge IRAScheduling(t)$
 20 $\quad \wedge UNCHANGED \langle TaskPri \rangle$

22 $PRATaskLocksResource(t, r) \triangleq$
 23 $\quad \wedge IRATaskLocksResource(t, r)$
 24 $\quad \wedge (ResState[r].WL = \{\}) \vee (\forall ti \in ResState[r].WL : TaskPri[t] \geq TaskPri[ti])$
 25 $\quad \wedge UNCHANGED \langle TaskPri \rangle$

27 $PRATaskReleasesResource(t, r) \triangleq$
 28 $\quad \wedge IRATaskReleasesResource(t, r)$
 29 $\quad \wedge UNCHANGED \langle TaskPri \rangle$

31 $PRATaskGoesToWL(t, r) \triangleq$
 32 $\quad \wedge MCIRATaskGoesToWL(t, r)$
 33 $\quad \wedge UNCHANGED \langle TaskPri \rangle$

35 $PRATaskGoesFromWLtoOwnership(t, r) \triangleq$
 36 $\quad \wedge IRATaskGoesFromWLtoOwnership(t, r)$
 37 $\quad \wedge \forall ti \in ResState[r].WL : TaskPri[t] \geq TaskPri[ti]$
 38 $\quad \wedge UNCHANGED \langle TaskPri \rangle$

40 $PRAPreemption(t) \triangleq$
 41 $\quad \wedge IRAPreemption(t)$
 42 $\quad \wedge UNCHANGED \langle TaskPri \rangle$

44 $PRANext \triangleq$
 45 $\quad \exists t \in Task :$
 46 $\quad \quad \vee \exists r \in Resource :$
 47 $\quad \quad \vee PRATaskLocksResource(t, r)$
 48 $\quad \quad \vee PRATaskReleasesResource(t, r)$
 49 $\quad \quad \vee PRATaskGoesToWL(t, r)$
 50 $\quad \quad \vee PRATaskGoesFromWLtoOwnership(t, r)$
 51 $\quad \vee PRAScheduling(t)$
 52 $\quad \vee PRAPreemption(t)$

54 $PRALive \triangleq$
 55 $\quad \forall t \in Task :$
 56 $\quad \quad \wedge \forall r \in Resource :$
 57 $\quad \quad \quad \wedge SF_{PRAvars}(PRATaskReleasesResource(t, r))$
 58 $\quad \quad \quad \wedge SF_{PRAvars}(PRATaskGoesFromWLtoOwnership(t, r))$
 59 $\quad \quad \wedge WF_{PRAvars}(PRAScheduling(t))$

61 $PRASpec \triangleq PRAInit \wedge \Box[PRANext]_{PRAvars}$

63 $PRALiveSpec \triangleq PRASpec \wedge PRALive$

64

```

65  $PRAHPRun \triangleq \forall t \in Task :$ 
66    $(\forall tj \in Task : TaskPri[t] \geq TaskPri[tj]) \Rightarrow (\Box \Diamond (TaskState[t] = \text{"Running"}))$ 

68  $PRAStarv \triangleq \forall t \in Task : \Diamond \Box (TaskState[t] \neq \text{"Running"}) \Rightarrow$ 
69    $\wedge \Diamond \Box (TaskOw[t].WaitsR \neq NoRes)$ 
70    $\wedge \Box \Diamond \exists tj \in Task \setminus \{t\} : \wedge (TaskPri[tj] > TaskPri[t])$ 
71    $\wedge \vee (TaskOw[tj].Owning = TaskOw[t].WaitsR)$ 
72    $\vee (TaskOw[tj].WaitsR = TaskOw[t].WaitsR)$ 

74  $PRARun \triangleq PRAHPRun \wedge PRAStarv$ 
75 ───────────────────────────────────────────────────────────────────────────────────
76 THEOREM  $PRASpec \Rightarrow \Box (PRATypeOk \wedge IRALock)$ 
77 THEOREM  $PRASpec \Rightarrow IRASpec$ 
78 THEOREM  $PRALiveSpec \Rightarrow PRARun$ 
79 ───────────────────────────────────────────────────────────────────────────────────

```

Figure 4.1: The *PrioResourceAccess* module.

A specification that implements the one in module *PrioResourceAccess* can be obtained by simply refining the module *SimpleResourceManagement* in a similar fashion. The result is module *PrioResourceManagement* shown in Figure 4.3.

Previously, in module *SimpleResourceManagement* (shown in Figure 3.13 on page 41) the two branches of action *SRMTaskRequestsResource* were not mutually exclusive. Now, in module *PrioResourceManagement*, the two branches of action *PRMTaskRequestsResource* are mutual exclusive. Variable *TaskPri* is used to eliminate the nondeterminism. Figure 4.2 depicts the conditions defined.

Res. Locked	Res. W.L.	Res. Notify	Highest priority	"Task Locks"	"Task goes to WL"
TRUE	---	---	---	✕	✓
FALSE	Empty	TRUE	TRUE	✕	✓
			FALSE	✕	✓
		FALSE	TRUE	✓	✕
			FALSE	✓	✕
	Not Empty	TRUE	TRUE	✕	✓
			FALSE	✕	✓
		FALSE	TRUE	✓	✕
			FALSE	✕	✓

Figure 4.2: Branches of action *PRMTaskRequestsResource*.

The module is not written as concisely as module *PrioResourceAccess* because the conditions expressed involving variable *TaskPri* have to be inserted inside the conditions of the IF THEN ELSE expressions. Also, to reduce the

number of modules created, the condition to avoid circular wait conditions is directly inserted in the module.¹

```

1  ┌────────────────────────── MODULE PrioResourceManagement ───────────────────┐
2  │ EXTENDS Naturals, FiniteSets, Sequences, SimpleResourceManagement          │
3  │                                                                              │
4  │ CONSTANTS PriMax                                                         │
5  │                                                                              │
6  │ VARIABLES TaskPri                                                         │
7  │ PRMvars  $\triangleq$   $\langle \text{ResState}, \text{TaskState}, \text{TaskOw}, \text{ResNotify}, \text{TaskPri} \rangle$  │
8  │                                                                              │
9  │ LOCAL INSTANCE PrioResourceAccess                                         │
10 │ LOCAL INSTANCE ResAllocGraph                                              │
11 │                                                                              │
12 │ PRMTypeOk  $\triangleq$                                                          │
13 │    $\wedge$  SRMTypeOk                                                         │
14 │    $\wedge \text{TaskPri} \in [\text{Task} \rightarrow 1 \dots \text{PriMax}]$                    │
15 │                                                                              │
16 │ PRMInit  $\triangleq$                                                          │
17 │    $\wedge$  SRMInit                                                         │
18 │    $\wedge \text{TaskPri} \in [\text{Task} \rightarrow 1 \dots \text{PriMax}]$                    │
19 │ ───────────────────────────────────────────────────────────────────────────┘
20 │ PRMScheduling(t)  $\triangleq$                                                          │
21 │    $\wedge$  SRMScheduling(t)                                                         │
22 │    $\wedge$  UNCHANGED  $\langle \text{TaskPri} \rangle$                                                          │
23 │                                                                              │
24 │ PRMTaskRequestsResource(t, r)  $\triangleq$                                                          │
25 │    $\wedge \text{TaskState}[t] = \text{"Running"}$                                                          │
26 │    $\wedge$  IF  $\wedge \text{ResState}[r].\text{Locked} = \text{FALSE}$                                                          │
27 │      $\wedge \text{ResNotify}[r] = \text{FALSE}$                                                          │
28 │      $\wedge \vee \text{ResState}[r].\text{WL} = \{\}$                                                          │
29 │      $\vee \forall ti \in \text{ResState}[r].\text{WL} : \text{TaskPri}[ti] \leq \text{TaskPri}[t]$                                                          │
30 │     THEN Task locks resource                                                         │
31 │        $\wedge \text{ResState}' = [\text{ResState} \text{ EXCEPT } ![r].\text{Locked} = \text{TRUE},$                                                          │
32 │          $![r].\text{Owner} = t]$                                                          │
33 │        $\wedge \text{TaskOw}' = [\text{TaskOw} \text{ EXCEPT } ![t].\text{Owning} = @ \cup \{r\}]$                                                          │
34 │        $\wedge$  UNCHANGED  $\langle \text{TaskState} \rangle$                                                          │
35 │     ELSE Task goes to WL                                                         │
36 │        $\wedge \neg(\text{ResState}[r].\text{Locked} = \text{TRUE} \wedge (\text{IsCyclicIn}(\langle t, r \rangle, \text{GraphLW}(\langle t, r \rangle))))$  │
37 │        $\wedge t \neq \text{ResState}[r].\text{Owner}$                                                          │
38 │        $\wedge \text{ResState}' = [\text{ResState} \text{ EXCEPT } ![r].\text{WL} = @ \cup \{t\}]$                                                          │
39 │        $\wedge \text{TaskState}' = [\text{TaskState} \text{ EXCEPT } ![t] = \text{"Waiting"}]$                                                          │
40 │        $\wedge \text{TaskOw}' = [\text{TaskOw} \text{ EXCEPT } ![t].\text{WaitsR} = r]$                                                          │
41 │        $\wedge$  UNCHANGED  $\langle \text{TaskPri}, \text{ResNotify} \rangle$                                                          │
42 │                                                                              │
43 │ PRMTaskReleasesResource(t, r)  $\triangleq$                                                          │
44 │    $\wedge$  SRMTaskReleasesResource(t, r)                                                         │
45 │    $\wedge$  UNCHANGED  $\langle \text{TaskPri} \rangle$                                                          │
46 │                                                                              │
47 │ PRMNotifyResourceUnlock(wokent, r)  $\triangleq$                                                          │
48 │    $\wedge \text{ResNotify}[r] = \text{TRUE}$                                                          │
49 │    $\wedge$  IF  $((\text{ResState}[r].\text{Locked} = \text{FALSE}) \wedge (\text{ResState}[r].\text{WL} \neq \{\}))$  │
50 │     THEN                                                         │
51 │        $\wedge \text{wokent} \in \text{ResState}[r].\text{WL}$                                                          │
52 │        $\wedge \forall tj \in \text{ResState}[r].\text{WL} : \text{TaskPri}[\text{wokent}] \geq \text{TaskPri}[tj]$ 

```

¹See formula 3.1 on page 35.

```

53       $\wedge ResState' = [ResState \text{ EXCEPT } ![r].Locked = \text{TRUE},$ 
54       $![r].Owner = woken,$ 
55       $![r].WL = @ \setminus \{woken\}]$ 
56       $\wedge TaskState' = [TaskState \text{ EXCEPT } ![woken] = \text{"Ready"}]$ 
57       $\wedge TaskOw' = [TaskOw \text{ EXCEPT } ![woken].WaitsR = NoRes,$ 
58       $![woken].Owning = @ \cup \{r\}]$ 
59      ELSE
60      UNCHANGED  $\langle ResState, TaskState, TaskOw \rangle$ 
61       $\wedge ResNotify' = [ResNotify \text{ EXCEPT } ![r] = \text{FALSE}]$ 
62       $\wedge$  UNCHANGED  $\langle TaskPri \rangle$ 

64   $PRMPreemption(t) \triangleq$ 
65       $\wedge SRMPreemption(t)$ 
66       $\wedge$  UNCHANGED  $\langle TaskPri \rangle$ 
67  ───────────────────────────────────────────────────────────────────────────────────
68   $PRMNext \triangleq$ 
69       $\exists t \in Task :$ 
70       $\vee \exists r \in Resource :$ 
71       $\vee PRMTaskRequestsResource(t, r)$ 
72       $\vee PRMTaskReleasesResource(t, r)$ 
73       $\vee PRMNotifyResourceUnlock(t, r)$ 
74       $\vee PRMScheduling(t)$ 
75       $\vee PRMPreemption(t)$ 

77   $PRMLive \triangleq$ 
78       $\forall t \in Task :$ 
79       $\wedge \forall r \in Resource :$ 
80       $\wedge SF_{PRMvars}(PRMTaskReleasesResource(t, r))$ 
81       $\wedge SF_{PRMvars}(PRMNotifyResourceUnlock(t, r))$ 
82       $\wedge WF_{PRMvars}(PRMScheduling(t))$ 

84   $PRMSpec \triangleq PRMInit \wedge \Box[PRMNext]_{PRMvars}$ 

86   $PRMLiveSpec \triangleq PRMSpec \wedge PRMLive$ 
87  ───────────────────────────────────────────────────────────────────────────────────
88   $PRMRun \triangleq PRARun$ 
89  ───────────────────────────────────────────────────────────────────────────────────
90  THEOREM  $PRMSpec \Rightarrow \Box(PRMTypOk \wedge SRMNotification \wedge IRALock)$ 
91  THEOREM  $PRMSpec \Rightarrow IRASpec \wedge SRMSpec \wedge PRASpec$ 
92  THEOREM  $PRMLiveSpec \Rightarrow PRMRun$ 
93  ───────────────────────────────────────────────────────────────────────────────────

```

Figure 4.3: The *PrioResourceManagement* module.

4.1.1 Did Fairness Get Killed?

In modules *InternalResourceAccess* and *SimpleResourceManagement*, described in chapter 3, we had defined the liveness condition

$$\forall t \in Task : \Box \Diamond (TaskState[t] = \text{"Running"}) \quad (4.1)$$

which asserted that every task would “run” infinitely often. Like that, the specification was said to be *fair*.

Property (4.1), however, is not satisfied by all behaviors of a system fulfilling the specification in modules *PrioResourceAccess* and *PrioResourceManagement*. This is because access to resources is granted based on the tasks' priorities. Like that, it is possible for a task to remain forever blocked on a resource that is successively locked by higher priority tasks.

The situation where fulfillment of a request is indefinitely postponed is called *starvation*. The question whether this is to be considered a problem of our specification is not trivial. Starvation would only happen because new requests of higher priority tasks would be always arriving. Assuming that higher priority requests should always be served first—after all, that's what priorities are for—the situation is unavoidable.²

Because specifications *PrioResourceAccess* and *PrioResourceManagement* allow behaviors that violate (4.1), this property is replaced by the less demanding

$$\begin{aligned} \forall t \in Task : (\forall tj \in Task : TaskPri[t] \geq TaskPri[tj]) \Rightarrow \\ \Box \Diamond (TaskState[t] = \text{"Running"}) \end{aligned} \quad (4.2)$$

defined in module *PrioResourceAccess* as property *PRAHPRun*. It is also defined the property *PRASurv* that asserts that if a task t eventually remains forever without running that implies that t is forever waiting on a resource r and that infinitely often there exists another task tj that has higher priority than t and that either owns resource r or is waiting for it.

Property *PRARun* is defined as the conjunction of *PRAHPRun* and *PRASurv*. In module *PrioResourceManagement*, property *PRMRun* is defined to equal *PRARun*.

4.1.2 The Priority Inversion Phenomenon

When concurrent tasks share resources in exclusive mode it is unavoidable that a task sometimes stays blocked for some time waiting for the release of a resource. This is true for any task, independently of its priority. Hence, higher priority tasks may be blocked by lower priority tasks. While unavoidable, such a blocking should be limited to the minimum length of time, i.e., the time strictly required for the lower priority task to execute its critical section and release the resource.

However, if no precautions are taken, it can happen that a higher priority task th stays blocked for much longer than the time needed by a lower priority task tl to release the resource. In fact, tl may be preempted by an (/ a set of) intermediate priority task(s) ti that prevents tl from executing its critical section. Consequently, ti indirectly blocks th . This is depicted in Figure 4.4. Such a scenario is called the *priority inversion phenomenon*.

²Naturally, if this is to be considered a problem it can be overcome by dynamically raising the blocked task's priority.

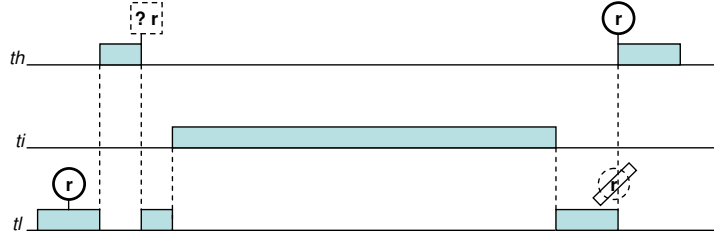


Figure 4.4: The priority inversion phenomenon.

4.2 The Priority Inheritance Protocol

The *Priority Inheritance Protocol*—developed for fixed priority based scheduling algorithms—offers a solution to the uncontrolled priority inversion phenomenon described in the previous section. Its basic principal is to modify the priority of tasks that block higher priority tasks. Like that, when a task tl blocks one or more higher-priority tasks it temporarily assumes (*inherits*) the highest priority of the tasks blocked by it. This prevents intermediate priority tasks from preempting tl and prolonging the blocking time of the higher priority tasks.

The protocol was first proposed in [28] and is also described, e.g., in the reference book [6]. These descriptions are, however, mainly textual. We will formalize the application of the protocol to the resource management mechanism developed in Chapter 3.

For that, it is necessary to modify the notion of task priority introduced in the modules *PrioResourceAccess* and *PrioResourceManagement* shown in Section 4.1. Since the protocol can modify the tasks' priorities, for every task, a fixed *nominal* priority and a dynamic *active* priority are distinguished. These will be represented, respectively, by variables $TNominalPri$ and $TActivePri$ in the TLA^+ models.

To start with, we will take the specification in module *InternalResourceAccess* (shown in Figure 3.6 on page 34) and extend it with the rules related to the priority inheritance protocol. These are:

- (i) When a task ti waits for a resource r , it transmits its active priority to the task tj that holds the resource. Task tj is said to inherit the priority of ti . In general, a task inherits the highest priority of the jobs blocked by it.
- (ii) When a task tj releases a resource r its active priority is updated as follows: if no other tasks are waiting for r , tj 's active priority is set to its nominal priority; otherwise it is set to the highest active priority of the jobs blocked by tj .
- (iii) Priority inheritance is transitive. That is, if a task ti blocks a task tj , and tj blocks tk , then ti inherits the priority of tk via tj .

In our model, these rules are materialized in the definition of actions *PIRATaskReleasesResource* and *PIRATaskGoesToWL*. The definition of actions *PIRATaskLocksResource* and *PIRATaskGoesFromWLtoOwnership* does not need to contain any computation related to the priority inheritance protocol. These two actions simply constrain the resource access to the highest (active) priority competing task, similarly to what was made in module *PrioResourceAccess*. The difference is that variable *TaskPri* is substituted by variable *TActivePri*.

The resulting specification is module *PInhResourceAccess* below.

```

1  ┌────────────────────────── MODULE PInhResourceAccess ───────────────────────────┐
2  │ EXTENDS Naturals, FiniteSets, MCInternalResourceAccess                      │
3  │                                                                              │
4  │ CONSTANTS PriMax                                                         │
5  │                                                                              │
6  │ VARIABLES TNominalPri, TActivePri                                         │
7  │ PIRAvars  $\triangleq$   $\langle \text{ResState}, \text{TaskState}, \text{TaskOw}, \text{TNominalPri}, \text{TActivePri} \rangle$  │
8  │                                                                              │
9  │ LOCAL INSTANCE ResAlocGraph                                              │
10 │                                                                              │
11 │ PIRATypeOk  $\triangleq$                                                          │
12 │    $\wedge$  IRATypeOk                                                         │
13 │    $\wedge$  TNominalPri  $\in [\text{Task} \rightarrow 1 \dots \text{PriMax}]$                  │
14 │    $\wedge$  TActivePri  $\in [\text{Task} \rightarrow 1 \dots \text{PriMax}]$                  │
15 │                                                                              │
16 │ PIRAInit  $\triangleq$                                                          │
17 │    $\wedge$  IRASInit                                                         │
18 │    $\wedge$  TNominalPri  $\in [\text{Task} \rightarrow 1 \dots \text{PriMax}]$                  │
19 │    $\wedge$  TActivePri  $= [t \in \text{Task} \mapsto \text{TNominalPri}[t]]$              │
20 │ ───────────────────────────────────────────────────────────────────────────────────┘
21 │ PrioRange  $\triangleq 1 \dots \text{PriMax}$                                        │
22 │                                                                              │
23 │ BlockT(t)  $\triangleq \{ \text{tup} \in \text{Resource} \times \text{Task} \times \text{PrioRange} : \wedge \text{tup}[1] \in \text{TaskOw}[t].\text{Owning}$  │
24 │    $\wedge \text{tup}[2] \in \text{ResState}[\text{tup}[1]].\text{WL}$  │
25 │    $\wedge \text{tup}[3] = \text{TActivePri}[\text{tup}[2]] \}$  │
26 │                                                                              │
27 │ BlockTExceptR(t, r)  $\triangleq \text{BlockT}(t) \setminus \{ e \in \text{BlockT}(t) : e[1] = r \}$  │
28 │ ───────────────────────────────────────────────────────────────────────────────────┘
29 │ PIRAScheduling(t)  $\triangleq$  │
30 │    $\wedge$  IRAScheduling(t) │
31 │    $\wedge$  UNCHANGED  $\langle \text{TNominalPri}, \text{TActivePri} \rangle$  │
32 │                                                                              │
33 │ PIRATaskLocksResource(t, r)  $\triangleq$  │
34 │    $\wedge$  IRATaskLocksResource(t, r) │
35 │    $\wedge (\text{ResState}[r].\text{WL} = \{ \}) \vee (\forall ti \in \text{ResState}[r].\text{WL} : \text{TActivePri}[ti] \leq \text{TActivePri}[t])$  │
36 │    $\wedge$  UNCHANGED  $\langle \text{TNominalPri}, \text{TActivePri} \rangle$  │
37 │                                                                              │
38 │ PIRATaskReleasesResource(t, r)  $\triangleq$  │
39 │    $\wedge$  IRATaskReleasesResource(t, r) │
40 │    $\wedge$  IF BlockTExceptR(t, r)  $\neq \{ \}$  │
41 │     THEN This “block” is related to the priority inheritance protocol │
42 │     LET │
43 │       hgPriTup  $\triangleq$  CHOOSE tup  $\in$  BlockTExceptR(t, r) : │
44 │          $\forall \text{otup} \in \text{BlockTExceptR}(t, r) : \text{tup}[3] \geq \text{otup}[3]$  │
45 │       hgPrio  $\triangleq$  hgPriTup[3] │

```

```

46   newActivePri  $\triangleq$  IF  $hgPri > TNominalPri[t]$ 
47   THEN  $hgPri$  ELSE  $TNominalPri[t]$ 
48   IN   Only  $t$  can have its active priority changed
49    $\wedge TActivePri' = [TActivePri \text{ EXCEPT } ![t] = newActivePri]$ 
50   ELSE
51      $TActivePri' = [TActivePri \text{ EXCEPT } ![t] = TNominalPri[t]]$ 
52    $\wedge \text{UNCHANGED } \langle TNominalPri \rangle$ 

54   PIRATaskGoesToWL( $t, r$ )  $\triangleq$ 
55    $\wedge MCIRATaskGoesToWL(t, r)$ 
56    $\wedge$  IF  $ResState[r].Locked = \text{TRUE}$ 
57   THEN   This “block” is related to the priority inheritance protocol
58   LET
59      $taskHoldingRes \triangleq ResState[r].Owner$ 
60   IN
61     IF  $TActivePri[t] > TActivePri[taskHoldingRes]$ 
62     THEN   Note that task  $t$  remains with the same priority
63     LET
64        $relTask \triangleq \{ti \in Task \setminus \{t\} : \text{Set of task that block task } t...$ 
65        $IsConnectedTo(ti, \langle t, r \rangle, GraphLW(\langle t, r \rangle))\}$ 
66        $lowerRelTask \triangleq \{tlow \in relTask : ... \text{and with lower priority}$ 
67        $TActivePri[t] > TActivePri[tlow]\}$ 
68        $newPri \triangleq TActivePri[t]$ 
69     IN
70        $TActivePri' = [ti \in Task \mapsto \text{IF } ti \in lowerRelTask$ 
71       THEN  $newPri$  ELSE  $TActivePri[ti]]$ 
72     ELSE
73       UNCHANGED  $\langle TActivePri \rangle$ 
74   ELSE
75     UNCHANGED  $\langle TActivePri \rangle$ 
76    $\wedge \text{UNCHANGED } \langle TNominalPri \rangle$ 

78   PIRATaskGoesFromWLtoOwnership( $t, r$ )  $\triangleq$ 
79    $\wedge IRATaskGoesFromWLtoOwnership(t, r)$ 
80    $\wedge \forall ti \in ResState[r].WL : TActivePri[t] \geq TActivePri[ti]$ 
81    $\wedge \text{UNCHANGED } \langle TNominalPri, TActivePri \rangle$ 

83   PIRAPreemption( $t$ )  $\triangleq$ 
84    $\wedge IRAPreemption(t)$ 
85    $\wedge \text{UNCHANGED } \langle TNominalPri, TActivePri \rangle$ 

```

```

87   PIRANext  $\triangleq$ 
88    $\exists t \in Task :$ 
89    $\vee \exists r \in Resource :$ 
90    $\vee PIRATaskLocksResource(t, r)$ 
91    $\vee PIRATaskReleasesResource(t, r)$ 
92    $\vee PIRATaskGoesToWL(t, r)$ 
93    $\vee PIRATaskGoesFromWLtoOwnership(t, r)$ 
94    $\vee PIRAScheduling(t)$ 
95    $\vee PIRAPreemption(t)$ 

97   PIRALive  $\triangleq$ 
98    $\forall t \in Task :$ 
99    $\wedge \forall r \in Resource :$ 
100    $\wedge SF_{PIRAvars}(PIRATaskReleasesResource(t, r))$ 
101    $\wedge SF_{PIRAvars}(PIRATaskGoesFromWLtoOwnership(t, r))$ 
102    $\wedge WF_{PIRAvars}(PIRAScheduling(t))$ 

```

Figure 4.5: The *PInhResourceAccess* module.

Figure 4.5: The *PInhResourceAccess* module.

Property *PIRAPrioInh* asserts that the priority inheritance concepts are correctly implemented. Property *PIRAPrioReset* asserts that the active priority of every task equals its nominal priority infinitely often. The *PIRARun* property is similar to property *PRARun*, presented in Section 4.1 and discussed in 4.1.1.

At the beginning of the module, it is found the definition of *BlockT(t)*. It is a state function that, for a task t , “returns” a set of tuples $\langle ri, ti, pi \rangle$ such that resource ri is owned by task t , task ti is waiting for ri , and pi is the active priority of ti . It is used to calculate every task’s supposed active priority, in the definition of invariant *PriosCorrect*.

The state function *BlockTExceptR(t,r)* “returns” the same set of tuples of *BlockT(t)* except those where ri equals r . It is used to update task t active priority when t releases resource r , with a *PIRATaskReleasesResource(t,r)* step. Note that, here, only the priority of task t can change. All the other tasks remain with the same priority.

In a *PIRATaskGoesToWL(t,r)* step several tasks can change of active priority. The task t that actually goes to the waiting list of resource r remains with the same priority. The tasks that change (raise) their active priority are the ones that directly or indirectly block task t . To identify such tasks, it is used the operator *IsConnectedTo*, defined in module *ResAlocGraph* (Figure 3.8, page 36).

To proceed, it is straightforward to apply the same concepts to the specification in module *SimpleResourceManagement* (in Figure 3.13 on page 41). The resulting specification is module *PIInResourceManagement* bellow.

```

1  ┌────────────────── MODULE PIInResourceManagement ───────────────────┐
2  EXTENDS Naturals, FiniteSets, SimpleResourceManagement
3
4  CONSTANTS PriMax
5
6  VARIABLES TNominalPri, TActivePri
7  PIRMvars  $\triangleq$   $\langle \text{ResState}, \text{TaskState}, \text{TaskOw}, \text{ResNotify}, \text{TNominalPri}, \text{TActivePri} \rangle$ 
8
9  LOCAL INSTANCE ResAlocGraph
10 LOCAL INSTANCE PIInResourceAccess
11
12 PIRMTypeOk  $\triangleq$ 
13    $\wedge$  SRMTypeOk
14    $\wedge$   $\text{TNominalPri} \in [\text{Task} \rightarrow 1 \dots \text{PriMax}]$ 
15    $\wedge$   $\text{TActivePri} \in [\text{Task} \rightarrow 1 \dots \text{PriMax}]$ 
16
17 PIRMInit  $\triangleq$ 
18    $\wedge$  SRMInit
19    $\wedge$   $\text{TNominalPri} \in [\text{Task} \rightarrow 1 \dots \text{PriMax}]$ 
20    $\wedge$   $\text{TActivePri} = [t \in \text{Task} \mapsto \text{TNominalPri}[t]]$ 
21 └──────────────────────────────────────────────────────────────────────────┘
22 PIRMScheduling(t)  $\triangleq$ 
23    $\wedge$  SRMScheduling(t)
24    $\wedge$  UNCHANGED  $\langle \text{TNominalPri}, \text{TActivePri} \rangle$ 
25
26 PIRMTaskRequestsResource(t, r)  $\triangleq$ 
27    $\wedge$   $\text{TaskState}[t] = \text{"Running"}$ 

```

```

28   $\wedge$  IF  $\wedge$   $ResState[r].Locked = FALSE$ 
29       $\wedge$   $ResNotify[r] = FALSE$ 
30       $\wedge$   $\vee$   $ResState[r].WL = \{\}$ 
31       $\vee \forall ti \in ResState[r].WL : TActivePri[ti] \leq TActivePri[t]$ 
32  THEN Task locks resource
33       $\wedge$   $ResState' = [ResState \text{ EXCEPT } ![r].Locked = TRUE,$ 
34           $![r].Owner = t]$ 
35       $\wedge$   $TaskOw' = [TaskOw \text{ EXCEPT } ![t].Owning = @ \cup \{r\}]$ 
36       $\wedge$  UNCHANGED  $\langle TaskState, TActivePri \rangle$ 
37  ELSE Task goes to WL
38       $\wedge \neg (ResState[r].Locked = TRUE \wedge (IsCyclicIn(\langle t, r \rangle, GraphLW(\langle t, r \rangle))))$ 
39       $\wedge t \neq ResState[r].Owner$ 
40       $\wedge$   $ResState' = [ResState \text{ EXCEPT } ![r].WL = @ \cup \{t\}]$ 
41       $\wedge$   $TaskState' = [TaskState \text{ EXCEPT } ![t] = \text{"Waiting"}]$ 
42       $\wedge$   $TaskOw' = [TaskOw \text{ EXCEPT } ![t].WaitsR = r]$ 
43       $\wedge$  IF  $ResState[r].Locked = TRUE$ 
44          THEN Equal to the "block" in action PIRATaskGoesToWL
45          LET
46               $taskHoldingRes \triangleq ResState[r].Owner$ 
47          IN
48              IF  $TActivePri[t] > TActivePri[taskHoldingRes]$ 
49                  THEN
50                      LET
51                           $relTask \triangleq \{ti \in Task \setminus \{t\} :$ 
52                               $IsConnectedTo(ti, \langle t, r \rangle, GraphLW(\langle t, r \rangle))\}$ 
53                           $lowerRelTask \triangleq \{tlow \in relTask :$ 
54                               $TActivePri[t] > TActivePri[tlow]\}$ 
55                           $newPri \triangleq TActivePri[t]$ 
56                      IN
57                           $TActivePri' = [ti \in Task \mapsto \text{IF } ti \in lowerRelTask$ 
58                               $\text{THEN } newPri \text{ ELSE } TActivePri[ti]]$ 
59                      ELSE
60                          UNCHANGED  $\langle TActivePri \rangle$ 
61                  ELSE
62                      UNCHANGED  $\langle TActivePri \rangle$ 
63           $\wedge$  UNCHANGED  $\langle ResNotify, TNominalPri \rangle$ 
64
65   $PIRMTaskReleasesResource(t, r) \triangleq$ 
66   $\wedge$   $SRMTaskReleasesResource(t, r)$ 
67   $\wedge$  IF  $BlockTExceptR(t, r) \neq \{\}$ 
68      THEN Equal to the "block" in action PIRATaskReleasesResource
69      LET
70           $hgPriTup \triangleq \text{CHOOSE } tup \in BlockTExceptR(t, r) :$ 
71               $\forall otup \in BlockTExceptR(t, r) : tup[3] \geq otup[3]$ 
72           $hgPrio \triangleq hgPriTup[3]$ 
73      IN
74          IF  $hgPrio > TNominalPri[t]$ 
75              THEN
76                   $TActivePri' = [TActivePri \text{ EXCEPT } ![t] = hgPrio]$ 
77              ELSE
78                   $TActivePri' = [TActivePri \text{ EXCEPT } ![t] = TNominalPri[t]]$ 
79          ELSE
80               $TActivePri' = [TActivePri \text{ EXCEPT } ![t] = TNominalPri[t]]$ 
81           $\wedge$  UNCHANGED  $\langle TNominalPri \rangle$ 
82
83   $PIRMNotifyResourceUnlock(wokent, r) \triangleq$ 
84   $\wedge$   $ResNotify[r] = TRUE$ 

```

```

85     $\wedge$  IF  $((ResState[r].Locked = FALSE) \wedge (ResState[r].WL \neq \{\}))$ 
86    THEN
87         $\wedge woken \in ResState[r].WL$ 
88         $\wedge \forall tj \in ResState[r].WL : TActivePri[woken] \geq TActivePri[tj]$ 
89         $\wedge ResState' = [ResState \text{ EXCEPT } ![r].Locked = TRUE,$ 
90             $![r].Owner = woken,$ 
91             $![r].WL = @ \setminus \{woken\}]$ 
92         $\wedge TaskState' = [TaskState \text{ EXCEPT } ![woken] = \text{"Ready"}]$ 
93         $\wedge TaskOw' = [TaskOw \text{ EXCEPT } ![woken].WaitsR = NoRes,$ 
94             $![woken].Owning = @ \cup \{r\}]$ 
95    ELSE
96         $\wedge$  UNCHANGED  $\langle ResState, TaskState, TaskOw \rangle$ 
97     $\wedge ResNotify' = [ResNotify \text{ EXCEPT } ![r] = FALSE]$ 
98     $\wedge$  UNCHANGED  $\langle TNominalPri, TActivePri \rangle$ 
100   $PIRMPreemption(t) \triangleq$ 
101     $\wedge SRMPreemption(t)$ 
102     $\wedge$  UNCHANGED  $\langle TNominalPri, TActivePri \rangle$ 
103  |-----|
104   $PIRMNext \triangleq$ 
105     $\exists t \in Task :$ 
106       $\vee \exists r \in Resource :$ 
107         $\vee PIRMTasksRequestsResource(t, r)$ 
108         $\vee PIRMTasksReleasesResource(t, r)$ 
109         $\vee PIRMNotifyResourceUnlock(t, r)$ 
110         $\vee PIRMScheduling(t)$ 
111         $\vee PIRMPreemption(t)$ 
113   $PIRMLive \triangleq$ 
114     $\forall t \in Task :$ 
115       $\wedge \forall r \in Resource :$ 
116         $\wedge SF_{PIRMvars}(PIRMTasksReleasesResource(t, r))$ 
117         $\wedge SF_{PIRMvars}(PIRMNotifyResourceUnlock(t, r))$ 
118         $\wedge WF_{PIRMvars}(PIRMScheduling(t))$ 
120   $PIRMSpec \triangleq PIRMInit \wedge \Box[PIRMNext]_{PIRMvars}$ 
122   $PIRMLiveSpec \triangleq PIRMSpec \wedge PIRMLive$ 
123  |-----|
124   $PIRMRun \triangleq PIRARun$ 
125   $PIRMPrioInh \triangleq PIRAPrioInh$ 
126   $PIRMPrioReset \triangleq PIRAPrioReset$ 
127  |-----|
128  THEOREM  $PIRMSpec \Rightarrow \Box(PIRMTypeOk \wedge SRMNotification \wedge IRALock \wedge PIRMPrioInh)$ 
129  THEOREM  $PIRMSpec \Rightarrow PIRASpec \wedge SRMSpec$ 
130  THEOREM  $PIRMLiveSpec \Rightarrow PIRMRun$ 
131  THEOREM  $PIRMLiveSpec \Rightarrow PIRMPrioReset$ 
132  |-----|

```

Figure 4.6: The *PInhResourceManagement* module.

4.3 Intermediate overview

Whether a specification is low- or high-level is a matter of perspective. Module *PInhResourceManagement* derives from modules *SimpleResourceManagement* and *PInhResourceAccess*, which in turn derive from previous modules. It can be considered, therefore, a relatively detailed low-level specification. It is the result of all the work that preceded it and that allowed solving many subtle details. Nevertheless, it does contain some “high-level” descriptions, namely in what concerns the application of the priority inheritance protocol. Indeed, the definitions of *IsConnectedTo* and *BlockTExceptR* are not straightforward to implement.

In the remaining of this chapter, we devise a possible implementation of the priority inheritance mechanisms. In the prosecution of this exercise, and because TLA^+ is such a powerful language, we decided to explore some of the consequences of distributing the implementation. It so happens that this exercise led us to find a conceptually very simple solution that can be applied both to distributed and to non-distributed systems.

The diagram below illustrates the modules presented thus far.

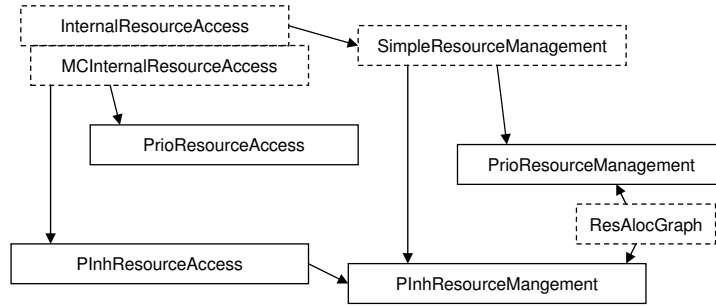


Figure 4.7: Modules built.

The description that follows gives clear evidence that the use of mathematics can be quite enlightening in the development of software systems.

4.4 Distributing the Priority Inheritance Computations

The NASA “Mars Pathfinder” that landed on Mars in 1997 experienced some problems that could have compromised all the mission. Fortunately, the problems were solved and the mission was considered successful. According to what was reported [24], the origin of the problem was a classical case of priority inversion. There was some resource being used without priority inheritance mechanisms enabled. Once the problem was diagnosed, it was possible to activate them, and the system behaved correctly from then on.

This intriguing description led to the aspiration of developing a solution where priority inheritance could be easily plugged or unplugged from the system. Recurring to conditional statements enabling and disabling the priority inheritance computations would be an obvious solution. Another possibility would be the use a separate task that would perform all the necessary calculations. In the end, and because TLA^+ is specially well suited for reasoning about such systems, it was decided to simulate a distributed application that could be implemented using two different nodes. The first node (say NA) controls the “normal” behavior of the system and the second node (say NB) performs the calculations needed to apply priority inheritance. The two nodes communicate by exchanging messages.

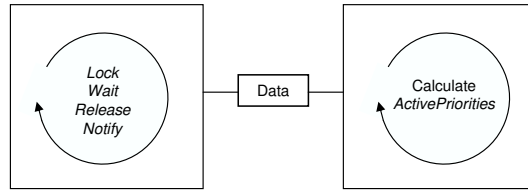


Figure 4.8: Distributed Priority Inheritance.

To start building the model, it is necessary to define which data the two nodes exchange. Node NA needs to send to node NB information related to the resource’s allocation. Node NB shall reply with the calculated active priorities of the tasks.

To allow node NB to form an image of the resource allocation, every “relevant” event is reported by the sent of a message. The relevant events are the times at witch a task t either (i) locks, (ii) waits on, or (iii) releases a resource r , and (iv) when a resource notification is performed. In TLA^+ , we will make each of these events correspond to the sent of a message $pitem$, where $pitem \in PInhItem$, and

$$PInhItem \triangleq [Type : \{ "TLocksR", "TWaitsOnR", "TReleasesR", "TIsNotified" \}, \\ OnTask : Task, \\ OnRes : Resource]^3$$

To represent the sent of such messages with some relative detail, we do not make them instantaneously “appear” in node NB . Instead, every message $pitem$ is added to a queue of messages $PGlobQ$ that hypothetically sits on node NA . The built of such queue is made in actions $DITaskRequestsResource$, $DITaskReleasesResource$, and $DINotifyUnlock$. The definition of any of these three actions is close to their “homonym” version of module $PInhResourceManagement$. The difference is that it is withdrawn the priority inheritance computation and is added the updating of variable $PGlobQ$.

³The correspondence between the (i–iv) events and the value of field $Type$ is obvious. Namely, (i) corresponds to “TLocksR”, (ii) to “TWaistsOnR”, (iii) to “TReleasesR”, and (iv) to “TIsNotified”.

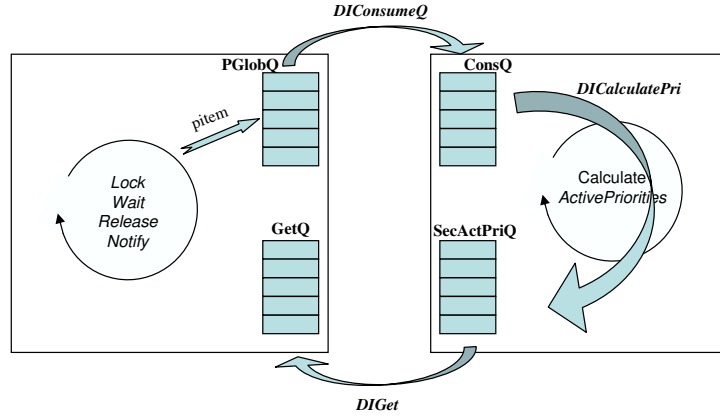


Figure 4.9: Distributed Priority Inheritance Computation.

The messages in *PGlobQ* are eventually sent to node *NB* and enqueued in *ConsQ*. This is represented by action *DIConsumeQ*. In this action, it is also updated the value of two new variables, namely *SecResState* and *SecTaskOw*. These are the variables used to form the image of the resource allocation, and their definition is similar to *ResState* and *TaskOw*, respectively. Except for the communication delay, these variables should contain the same information. Whenever there is no delay their value should be exactly the same. This statement is formally expressed by property *DISecCorrect*:

$$\begin{aligned}
 DISecCorrect &\triangleq \\
 &Len(PGlobQ) = 0 \Rightarrow \\
 &\quad \wedge ResState = SecResState \\
 &\quad \wedge TaskOw = SecTaskOw
 \end{aligned}$$

The values of the tasks' active priorities are calculated in action *DICalculatePri*. Here, the messages stored in *ConsQ* are individually consumed and the calculation is performed as follows: if the event registered in the message was either a (i) or (iv) event no changes are made. If it was a (ii) or (iii) event then the tasks priorities are calculated exactly the same way as made before in module *PInhResourceManagement* (Figure 4.6, page 59). The only difference is that variables *SecResState* and *SecTaskOw* are used in place of *ResState* and *TaskOw*, respectively. The result of the calculation updates variable *SecTActivePri* and is enqueued in *SecActPriQ*, to be sent to node *NA*.⁴

Action *DIGet* represents the sent of the messages stored in *SecActPriQ*, hypothetically sitting on node *NB*, to *GetQ*, hypothetically sitting on node *NA*.

Whenever there are messages in *GetQ*, they can be consumed and the effective value of every task's active priority is updated. This is represented by

⁴In the definition of action *DICalculatePri*, all the four branches are clearly mutually exclusive. Therefore, the CASE operator can be used with no reserves. For more comments on the use of CASE statements see Section 2.2.3 on page 21.

```

1  | MODULE DistributedInheritance
2  | EXTENDS Naturals, FiniteSets, Sequences, TLC, PInhResourceManagement

4  CONSTANTS QMax

6  VARIABLES PGlobQ, ConsQ, SecResState, SecTaskOw, SecTActivePri, SecActPriQ, GetQ

8  DIVars  $\triangleq$   $\langle ResState, TaskState, TaskOw, ResNotify, TNominalPri, TActivePri,$ 
9             PGlobQ, ConsQ, SecResState, SecTaskOw, SecTActivePri, SecActPriQ, GetQ  $\rangle$ 

11 LOCAL INSTANCE SecResAlocGraph
12 LOCAL INSTANCE Blockings
13 LOCAL INSTANCE InternalResourceAccess

15 LOCAL Perms  $\triangleq$  Permutations(Task)  $\cup$  Permutations(Resource)

17 PInhItem  $\triangleq$  [Type : { "TLocksR", "TWaitsOnR", "TReleasesR", "TIsNotified" },
18              OnTask : Task,
19              OnRes : Resource]

21 DITypeOk  $\triangleq$ 
22    $\wedge$  PIRMTypeOk
23    $\wedge$  PGlobQ  $\in Seq(PInhItem)$ 
24    $\wedge$  ConsQ  $\in Seq(PInhItem)$ 
25    $\wedge$  SecResState  $\in [Resource \rightarrow [Locked : BOOLEAN,$ 
26                                   WL : SUBSET (Task),
27                                   Owner : Task  $\cup$  { NoTask }]]
28    $\wedge$  SecTaskOw  $\in [Task \rightarrow [Owning : SUBSET (Resource),$ 
29                                   WaitsR : Resource  $\cup$  { NoRes }]]
30    $\wedge$  SecTActivePri  $\in [Task \rightarrow 1 \dots PriMax]$ 
31    $\wedge$  SecActPriQ  $\in Seq([Task \rightarrow 1 \dots PriMax])$ 
32    $\wedge$  GetQ  $\in Seq([Task \rightarrow 1 \dots PriMax])$ 

34 DIInit  $\triangleq$ 
35    $\wedge$  PIRMInit
36    $\wedge$  PGlobQ =  $\langle \rangle$ 
37    $\wedge$  ConsQ =  $\langle \rangle$ 
38    $\wedge$  SecResState =  $[r \in Resource \mapsto [Locked \mapsto FALSE,$ 
39                                   WL  $\mapsto \{ \}$ ,
40                                   Owner  $\mapsto NoTask]]$ 
41    $\wedge$  SecTaskOw =  $[t \in Task \mapsto [Owning \mapsto \{ \},$ 
42                                   WaitsR  $\mapsto NoRes]]$ 
43    $\wedge$  SecTActivePri =  $[t \in Task \mapsto TNominalPri[t]]$ 
44    $\wedge$  SecActPriQ =  $\langle \rangle$ 
45    $\wedge$  GetQ =  $\langle \rangle$ 

47 |-----
48 DIScheduling(t)  $\triangleq$ 
49    $\wedge$  PIRMScheduling(t)
50    $\wedge$  UNCHANGED  $\langle PGlobQ, ConsQ, SecResState, SecTaskOw,$ 
51               SecTActivePri, SecActPriQ, GetQ  $\rangle$ 

53 DITaskRequestsResource(t, r)  $\triangleq$ 

```

```

54   $\wedge TaskState[t] = \text{"Running"}$ 
55   $\wedge Len(PGlobQ) < QMax$ 
56   $\wedge IF \wedge ResState[r].Locked = FALSE$ 
57     $\wedge ResNotify[r] = FALSE$ 
58     $\wedge \vee ResState[r].WL = \{\}$ 
59     $\vee \forall ti \in ResState[r].WL : TActivePri[ti] \leq TActivePri[t]$ 
60  THEN Task locks resource
61     $\wedge ResState' = [ResState \text{ EXCEPT } ![r].Locked = TRUE,$ 
62       $![r].Owner = t]$ 
63     $\wedge TaskOw' = [TaskOw \text{ EXCEPT } ![t].Owning = @ \cup \{r\}]$ 
64     $\wedge LET$ 
65       $pitem \triangleq \text{CHOOSE } pi \in PInhItem : \wedge pi.Type = \text{"TLocksR"} \quad (i)$ 
66       $\wedge pi.OnTask = t$ 
67       $\wedge pi.OnRes = r$ 
68    IN
69     $PGlobQ' = Append(PGlobQ, pitem)$ 
71   $\wedge \text{UNCHANGED } \langle TaskState, TActivePri \rangle$ 
72  ELSE Task goes to WL
73     $\wedge \neg(ResState[r].Locked = TRUE \wedge (IsCyclicIn(\langle t, r \rangle, GraphLW(\langle t, r \rangle))))$ 
74     $\wedge t \neq ResState[r].Owner$ 
75     $\wedge ResState' = [ResState \text{ EXCEPT } ![r].WL = @ \cup \{t\}]$ 
76     $\wedge TaskState' = [TaskState \text{ EXCEPT } ![t] = \text{"Waiting"}]$ 
77     $\wedge TaskOw' = [TaskOw \text{ EXCEPT } ![t].WaitsR = r]$ 
78     $\wedge LET$ 
79       $pitem \triangleq \text{CHOOSE } pi \in PInhItem : \wedge pi.Type = \text{"TWaitsOnR"} \quad (ii)$ 
80       $\wedge pi.OnTask = t$ 
81       $\wedge pi.OnRes = r$ 
82    IN
83     $PGlobQ' = Append(PGlobQ, pitem)$ 
84   $\wedge \text{UNCHANGED } \langle ResNotify, TNominalPri \rangle$ 
85   $\wedge \text{UNCHANGED } \langle TActivePri \rangle$ 
86   $\wedge \text{UNCHANGED } \langle ConsQ, SecResState, SecTaskOw, SecTActivePri, SecActPriQ, GetQ \rangle$ 
88   $DITaskReleasesResource(t, r) \triangleq$ 
89     $\wedge SRMTaskReleasesResource(t, r)$ 
90     $\wedge Len(PGlobQ) < QMax$ 
91     $\wedge LET$ 
92       $pitem \triangleq \text{CHOOSE } pi \in PInhItem : \wedge pi.Type = \text{"TReleasesR"} \quad (iii)$ 
93       $\wedge pi.OnTask = t$ 
94       $\wedge pi.OnRes = r$ 
95    IN
96     $PGlobQ' = Append(PGlobQ, pitem)$ 
98   $\wedge \text{UNCHANGED } \langle TNominalPri, TActivePri \rangle$ 
99   $\wedge \text{UNCHANGED } \langle ConsQ, SecResState, SecTaskOw, SecTActivePri, SecActPriQ, GetQ \rangle$ 
101  $DINotifyResourceUnlock(wokent, r) \triangleq$ 
102   $\wedge ResNotify[r] = TRUE$ 
103   $\wedge IF ((ResState[r].Locked = FALSE) \wedge (ResState[r].WL \neq \{\}))$ 
104  THEN
105     $\wedge Len(PGlobQ) < QMax$ 
106     $\wedge wokent \in ResState[r].WL$ 
107     $\wedge \forall tj \in ResState[r].WL : TActivePri[wokent] \geq TActivePri[tj]$ 
108     $\wedge ResState' = [ResState \text{ EXCEPT } ![r].Locked = TRUE,$ 
109       $![r].Owner = wokent,$ 
110       $![r].WL = @ \setminus \{wokent\}]$ 
111     $\wedge TaskState' = [TaskState \text{ EXCEPT } ![wokent] = \text{"Ready"}]$ 

```

4.4. DISTRIBUTING THE PRIORITY INHERITANCE COMPUTATIONS 65

```

112       $\wedge TaskOw' = [TaskOw \text{ EXCEPT } ![woken].WaitsR = NoRes,$ 
113       $![woken].Owning = @ \cup \{r\}]$ 
114       $\wedge \text{LET}$ 
115       $pitem \triangleq \text{CHOOSE } pi \in PInhItem : \wedge pi.Type = \text{"TIsNotified"} \quad (iv)$ 
116       $\wedge pi.OnTask = woken$ 
117       $\wedge pi.OnRes = r$ 
118       $\text{IN}$ 
119       $PGlobQ' = Append(PGlobQ, pitem)$ 
121       $\text{ELSE}$ 
122       $\wedge \text{UNCHANGED } \langle ResState, TaskState, TaskOw, PGlobQ \rangle$ 
123       $\wedge ResNotify' = [ResNotify \text{ EXCEPT } ![r] = FALSE]$ 
124       $\wedge \text{UNCHANGED } \langle TNominalPri, TActivePri \rangle$ 
125       $\wedge \text{UNCHANGED } \langle ConsQ, SecResState, SecTaskOw, SecTActivePri, SecActPriQ, GetQ \rangle$ 
127       $DIPreemption(t) \triangleq$ 
128       $\wedge PIRMPreemption(t)$ 
129       $\wedge \text{UNCHANGED } \langle PGlobQ, ConsQ, SecResState, SecTaskOw,$ 
130       $SecTActivePri, SecActPriQ, GetQ \rangle$ 
131       $\mid$ 
132       $DIConsumeQ \triangleq$ 
133       $\wedge Len(PGlobQ) > 0$ 
134       $\wedge Len(ConsQ) < QMax$ 
135       $\wedge \text{LET}$ 
136       $type \triangleq PGlobQ[1].Type$ 
137       $t \triangleq PGlobQ[1].OnTask$ 
138       $r \triangleq PGlobQ[1].OnRes$ 
139       $\text{IN}$ 
140       $\wedge \text{CASE } type = \text{"TLocksR"} \rightarrow$ 
141       $\wedge SecResState' = [SecResState \text{ EXCEPT } ![r].Locked = TRUE,$ 
142       $![r].Owner = t]$ 
143       $\wedge SecTaskOw' = [SecTaskOw \text{ EXCEPT } ![t].Owning = @ \cup \{r\}]$ 
145       $\square type = \text{"TWaitsOnR"} \rightarrow$ 
146       $\wedge SecResState' = [SecResState \text{ EXCEPT } ![r].WL = @ \cup \{t\}]$ 
147       $\wedge SecTaskOw' = [SecTaskOw \text{ EXCEPT } ![t].WaitsR = r]$ 
149       $\square type = \text{"TReleasesR"} \rightarrow$ 
150       $\wedge SecResState' = [SecResState \text{ EXCEPT } ![r].Owner = NoTask,$ 
151       $![r].Locked = FALSE]$ 
152       $\wedge SecTaskOw' = [SecTaskOw \text{ EXCEPT } ![t].Owning = @ \setminus \{r\}]$ 
154       $\square type = \text{"TIsNotified"} \rightarrow$ 
155       $\wedge SecResState' = [SecResState \text{ EXCEPT } ![r].Locked = TRUE,$ 
156       $![r].Owner = t,$ 
157       $![r].WL = @ \setminus \{t\}]$ 
158       $\wedge SecTaskOw' = [SecTaskOw \text{ EXCEPT } ![t].WaitsR = NoRes,$ 
159       $![t].Owning = @ \cup \{r\}]$ 
161       $\wedge ConsQ' = Append(ConsQ, PGlobQ[1])$ 
162       $\wedge PGlobQ' = Tail(PGlobQ)$ 
163       $\wedge \text{UNCHANGED } \langle ResState, TaskState, TaskOw, ResNotify, TNominalPri,$ 
164       $TActivePri, SecTActivePri, SecActPriQ, GetQ \rangle$ 
166       $DICalculatePri \triangleq$ 
167       $\wedge Len(ConsQ) > 0$ 
168       $\wedge \text{LET}$ 
169       $type \triangleq ConsQ[1].Type$ 

```

```

170    $t \triangleq \text{ConsQ}[1].\text{OnTask}$ 
171    $r \triangleq \text{ConsQ}[1].\text{OnRes}$ 
172   IN
173    $\wedge \text{CASE } \text{type} = \text{"TLocksR"} \rightarrow \text{(i)}$ 
174     UNCHANGED  $\langle \text{SecTActivePri}, \text{SecActPriQ} \rangle$ 

176    $\square \text{type} = \text{"TWaitsOnR"} \rightarrow \text{(ii)}$ 
177     IF  $\text{SecResState}[r].\text{Locked} = \text{TRUE}$ 
178       THEN
179         LET
180            $\text{taskHoldingRes} \triangleq \text{SecResState}[r].\text{Owner}$ 
181         IN
182           IF  $\text{SecTActivePri}[t] > \text{SecTActivePri}[\text{taskHoldingRes}]$ 
183             THEN Similar to the "block" in action PIRATaskGoesToWL,
184                  but using "Sec-definitions".
185                  $\text{relTask} \triangleq \{ti \in \text{Task} \setminus \{t\} :$ 
186                    $\text{IsConnectedTo}(ti, \langle t, r \rangle, \text{SecGraphLW}(\langle t, r \rangle))\}$ 
187                  $\text{lowerRelTask} \triangleq \{tlow \in \text{relTask} :$ 
188                    $\text{SecTActivePri}[t] > \text{SecTActivePri}[tlow]\}$ 
189                  $\text{newPri} \triangleq \text{SecTActivePri}[t]$ 
190             IN
191                $\wedge \text{Len}(\text{SecActPriQ}) < QMax$ 
192                $\wedge \text{SecTActivePri}' = [ti \in \text{Task} \mapsto \text{IF } ti \in \text{lowerRelTask}$ 
193                 THEN  $\text{newPri}$  ELSE  $\text{SecTActivePri}[ti]$ ]
194                $\wedge \text{SecActPriQ}' = \text{Append}(\text{SecActPriQ}, \text{SecTActivePri}')$ 
195             ELSE
196               UNCHANGED  $\langle \text{SecTActivePri}, \text{SecActPriQ} \rangle$ 
197           ELSE
198             UNCHANGED  $\langle \text{SecTActivePri}, \text{SecActPriQ} \rangle$ 

200    $\square \text{type} = \text{"TReleasesR"} \rightarrow \text{(iii)}$ 
201      $\wedge \text{IF } \text{SecBlockTExceptR}(t, r) \neq \{\}$ 
202       THEN Similar to the "block" in action PIRATaskReleasesResource,
203            but using "Sec-definitions".
204             $\text{hgPriTup} \triangleq \text{CHOOSE } \text{tup} \in \text{SecBlockTExceptR}(t, r) :$ 
205               $\forall \text{otup} \in \text{SecBlockTExceptR}(t, r) : \text{tup}[3] \geq \text{otup}[3]$ 
206             $\text{hgPrio} \triangleq \text{hgPriTup}[3]$ 
207          IN
208            IF  $\text{hgPrio} > \text{TNominalPri}[t]$ 
209              THEN
210                 $\text{SecTActivePri}' = [\text{SecTActivePri} \text{ EXCEPT } ![t] = \text{hgPrio}]$ 
211              ELSE
212                 $\text{SecTActivePri}' = [\text{SecTActivePri} \text{ EXCEPT } ![t] = \text{TNominalPri}[t]]$ 
213            ELSE
214               $\text{SecTActivePri}' = [\text{SecTActivePri} \text{ EXCEPT } ![t] = \text{TNominalPri}[t]]$ 
215             $\wedge \text{Len}(\text{SecActPriQ}) < QMax$ 
216             $\wedge \text{SecActPriQ}' = \text{Append}(\text{SecActPriQ}, \text{SecTActivePri}')$ 

218    $\square \text{type} = \text{"TIsNotified"} \rightarrow \text{(iv)}$ 
219     UNCHANGED  $\langle \text{SecTActivePri}, \text{SecActPriQ} \rangle$ 
220    $\wedge \text{ConsQ}' = \text{Tail}(\text{ConsQ})$ 
221    $\wedge \text{UNCHANGED } \langle \text{ResState}, \text{TaskState}, \text{TaskOw}, \text{TNominalPri}, \text{TActivePri},$ 
222      $\text{ResNotify}, \text{PGlobQ}, \text{SecResState}, \text{SecTaskOw}, \text{GetQ} \rangle$ 

224    $DIGet \triangleq$ 
225      $\wedge \text{Len}(\text{SecActPriQ}) > 0$ 
226      $\wedge \text{Len}(\text{GetQ}) < QMax$ 

```

4.4. DISTRIBUTING THE PRIORITY INHERITANCE COMPUTATIONS 67

```

227       $\wedge \text{GetQ}' = \text{Append}(\text{GetQ}, \text{SecActPriQ}[1])$ 
228       $\wedge \text{SecActPriQ}' = \text{Tail}(\text{SecActPriQ})$ 
229       $\wedge \text{UNCHANGED} \langle \text{ResState}, \text{TaskState}, \text{TaskOw}, \text{TNominalPri}, \text{TActivePri},$ 
230       $\text{ResNotify}, \text{PGlobQ}, \text{ConsQ}, \text{SecResState}, \text{SecTaskOw}, \text{SecTActivePri} \rangle$ 

232   $\text{DIModifyPri} \triangleq$ 
233       $\wedge \text{Len}(\text{GetQ}) > 0$ 
234       $\wedge \text{TActivePri}' = \text{GetQ}[1]$ 
235       $\wedge \text{GetQ}' = \text{Tail}(\text{GetQ})$ 
236       $\wedge \text{UNCHANGED} \langle \text{ResState}, \text{TaskState}, \text{TaskOw}, \text{TNominalPri}, \text{ResNotify}, \text{PGlobQ},$ 
237       $\text{ConsQ}, \text{SecResState}, \text{SecTaskOw}, \text{SecTActivePri}, \text{SecActPriQ} \rangle$ 
238  |
239   $\text{DISimpleNext} \triangleq$ 
240       $\exists t \in \text{Task} :$ 
241           $\vee \exists r \in \text{Resource} :$ 
242               $\vee \text{DITaskRequestsResource}(t, r)$ 
243               $\vee \text{DITaskReleasesResource}(t, r)$ 
244               $\vee \text{DINotifyResourceUnlock}(t, r)$ 
245               $\vee \text{DIScheduling}(t)$ 
246               $\vee \text{DIPreemption}(t)$ 

248   $\text{DIPInhNext} \triangleq \text{DIconsumeQ} \vee \text{DICalculatePri} \vee \text{DIGet} \vee \text{DIModifyPri}$ 

250   $\text{DINext} \triangleq$ 
251       $\vee \text{DISimpleNext}$ 
252       $\vee \text{DIPInhNext}$ 

254   $\text{DILive} \triangleq$ 
255       $\forall t \in \text{Task} :$ 
256           $\wedge \forall r \in \text{Resource} :$ 
257               $\wedge \text{SF}_{\text{DIvars}}(\text{DITaskReleasesResource}(t, r))$ 
258               $\wedge \text{SF}_{\text{DIvars}}(\text{DINotifyResourceUnlock}(t, r))$ 
259               $\wedge \text{WF}_{\text{DIvars}}(\text{DIScheduling}(t))$ 
260               $\wedge \text{WF}_{\text{DIvars}}(\text{DIPInhNext})$ 

262   $\text{DISpec} \triangleq \text{DIInit} \wedge \Box[\text{DINext}]_{\text{DIvars}}$ 

264   $\text{DILiveSpec} \triangleq \text{DISpec} \wedge \text{DILive}$ 
265  |
266   $\text{DIPrioInh} \triangleq$ 
267       $(\forall q \in \{\text{PGlobQ}, \text{ConsQ}, \text{SecActPriQ}, \text{GetQ}\} : \text{Len}(q) = 0) \Rightarrow \text{PIRMPrioInh}$ 

269   $\text{DISEcCorrect} \triangleq$ 
270       $\text{Len}(\text{PGlobQ}) = 0 \Rightarrow$ 
271           $\wedge \text{ResState} = \text{SecResState}$ 
272           $\wedge \text{TaskOw} = \text{SecTaskOw}$ 
273  |
274   $\text{DIRun} \triangleq \text{PIRMRun}$ 
275  |
276   $\text{DIPrioReset} \triangleq \text{PIRMPrioReset}$ 
277  |

```

Figure 4.10: The *DistributedInheritance* module.

Module *DistributedInheritance* instantiates modules *SecResAlocGraph* and *Blockings*. Module *SecResAlocGraph* contains the definitions necessary to build a resource allocation graph as described in Section 3.3.1, but using variables *SecResState* and *SecTaskOw* instead of *ResState* and *TaskOw*, respectively. This is necessary for the application of the operator *IsConnectedTo* in the branch *type* = “*TWaitsOnR*” of the CASE expression in action *DICalculatePri*.

Module *Blockings* contains the definitions of the state functions *SecBlockT* and *SecBlockTExceptR*. These are similar to *BlockT* and *BlocTExceptR*, except for the fact that variables *SecResState* and *SecTaskOw* are used instead of variables *ResState* and *TaskOw*, respectively. The state function *SecBlockTExceptR* is used in the branch *type* = “*TReleasesR*” of the CASE expression in action *DICalculatePri*.

Modules *SecResAlocGraph* and *Blockings* are shown in Figures 4.11 and 4.12, respectively. Naturally, it would not be necessary to make such definitions in separate modules. Spreading a specification over different modules is a matter of taste.

```

1  ┌────────────────────────── MODULE SecResAlocGraph ───────────────────┐
2  EXTENDS ResAlocGraph
3
4  VARIABLES SecResState, SecTaskOw
5
6  SecNodeL   $\triangleq$  {  $n \in \text{Resource} \times \text{Task} :$ 
7                     $\wedge \text{SecResState}[n[1]].\text{Locked} = \text{TRUE}$ 
8                     $\wedge n[2] = \text{SecResState}[n[1]].\text{Owner}$  }
9
10 SecNodeW  $\triangleq$  {  $n \in \text{Task} \times \text{Resource} : \text{SecTaskOw}[n[1]].\text{WaitsR} = n[2]$  }
11
12 SecNodeLW(fut)  $\triangleq$  SecNodeL  $\cup$  SecNodeW  $\cup$  {fut}
13
14 SecGraphLW(fut)  $\triangleq$  [ node  $\mapsto \text{SecNodeLW}(fut),$ 
15                    edge  $\mapsto \text{EdgeLW}(\text{SecNodeLW}(fut))$  ]
16 └────────────────────────────────────────────────────────────────────────┘

```

Figure 4.11: The *SecResAlocGraph* module.

At the end of module *DistributedInheritance*, the invariant *DIPrioInh* is specified. It asserts that whenever all the listed queues are empty *PIRMPrioInh* must be true.⁵ This represents that whenever there are no messages in transit, i.e., there are no computations related to the priority inheritance yet to perform, then every task must have its active priority in accordance with the priority inheritance protocol laws.

The temporal properties *DIRun* and *DIPrioReset* directly import the definitions in module *PInhResourceManagement*.

No theorems are specified about this specification. In fact, there is a problem with the system. This is addressed in the next section.

⁵The definition of *PIRMPrioInh* is imported from module *PInhResourceManagemt* (Figure 4.6 on page 59), which in turn imports the definition of *PIRAPrioInh* made in module *PInhResourceAccess* (Figure 4.5 on page 56).

4.4. DISTRIBUTING THE PRIORITY INHERITANCE COMPUTATIONS 69

```

1  ┌────────────────────────── MODULE Blockings ───────────────────────────┐
2  EXTENDS Naturals, Sequences
3
4  CONSTANTS PriMax, Resource, Task
5
6  VARIABLES SecTActivePri, SecTaskOw, SecResState
7
8  PrioRange  $\triangleq$  1 .. PriMax
9
10 SecBlockT(t)  $\triangleq$ 
11   { tup  $\in$  Resource  $\times$  Task  $\times$  PrioRange :  $\wedge$  tup[1]  $\in$  SecTaskOw[t].Owning
12      $\wedge$  tup[2]  $\in$  SecResState[tup[1]].WL
13      $\wedge$  tup[3] = SecTActivePri[tup[2]] }
14
15 SecBlockTExceptR(t, r)  $\triangleq$  SecBlockT(t)  $\setminus$  { e  $\in$  SecBlockT(t) : e[1] = r }
16 └──────────────────────────────────────────────────────────────────────────┘

```

Figure 4.12: The *Blockings* module.

4.4.1 A Source of Problems

Model checking the specification in module *DistributedInheritance* TLC reports a violation of the invariant *DIPrioInh*. It takes two tasks of different nominal priorities and one resource to demonstrate the error, which is found after 22 steps.⁶ Analyzing the error trace, it is clear that the problem is due to the delay in the actualization of the tasks' active priorities. Should this delay disappear and no problem would have been found. Is this so? Can we be sure that the referred delay is the only source of problems in the specification? Yes, we can, it takes almost no more than the addition of a single line expression in the definition of the next-state relation of module *DistributedInheritance* to evaluate this statement.

In order to leave module *DistributedInheritance* unchanged, a new specification is created. This is module *SyncInheritance* in Figure 4.13. It extends module *DistributedInheritance* and constrains the behavior in a very simple way: whenever either a task (i) locks, (ii) waists on, or (iii) releases a resource, or (iv) a resource if notified, which causes the appearance of a message in *PGlobQ*, all the computations related to the possible actualization of the tasks priorities must be performed before any other step may occur.

The behavior restriction is materialized in the definition of the next-state relation *SINext*:

$$\begin{aligned}
 SINext &\triangleq \\
 &\vee \wedge \textit{CalculatingFlag} = \text{FALSE} \\
 &\quad \wedge \textit{SISimpleNext} \\
 &\vee \textit{SIPinhNext}
 \end{aligned}$$

where *SISimpleNext* is the disjunction of all the actions that are not directly related to the computation of the tasks priorities, *SIPinhNext* is the disjunction

⁶Appendix A shows the configuration file used and the error trace reported by TLC.

of the actions that perform the referred computation, and *CalculatingFlag* is a state function that is true whenever one of the listed queues is not empty:

$$\begin{aligned} \text{CalculatingFlag} &\triangleq \\ &\exists q \in \{PGlobQ, ConsQ, SecActPriQ, GetQ\} : Len(q) > 0 \end{aligned}$$

```

1  ┌────────────────────────── MODULE SyncInheritance ───────────────────┐
2  EXTENDS DistributedInheritance
3  SIvars  $\triangleq$  DIvars
4  LOCAL INSTANCE InternalResourceAccess
5  LOCAL Perms  $\triangleq$  Permutations(Task)  $\cup$  Permutations(Resource)
6  SITypeOk  $\triangleq$  DITypeOk
7  SIInit  $\triangleq$  DIInit
8  ┌────────────────────────────────────────────────────────────────────────┐
9  CalculatingFlag  $\triangleq$ 
10      $\exists q \in \{PGlobQ, ConsQ, SecActPriQ, GetQ\} : Len(q) > 0$ 
11 └────────────────────────────────────────────────────────────────────────┘
12 SISimpleNext  $\triangleq$  DISimpleNext
14 SIPInhNext  $\triangleq$  DIPInhNext
16 SINext  $\triangleq$ 
17      $\vee \wedge \text{CalculatingFlag} = \text{FALSE}$ 
18      $\wedge$  SISimpleNext
19      $\vee$  SIPInhNext
21 SILive  $\triangleq$  DILive
23 SISpec  $\triangleq$  SIInit  $\wedge \Box[SINext]_{SIvars}$ 
25 SILiveSpec  $\triangleq$  SISpec  $\wedge$  SILive
26 ┌────────────────────────────────────────────────────────────────────────┐
27 SIPrioInh  $\triangleq$  DIPrioInh
28 SISecCorrect  $\triangleq$  DISecCorrect
29 ┌────────────────────────────────────────────────────────────────────────┐
30 SIRun  $\triangleq$  DIRun
31 SIPrioReset  $\triangleq$  DIPrioReset
32 ┌────────────────────────────────────────────────────────────────────────┐
33 THEOREM SISpec  $\Rightarrow \Box(SITypeOk \wedge SRMNotification \wedge IRALock \wedge SIPrioInh \wedge SISecCorrect)$ 
34 THEOREM SISpec  $\Rightarrow SRMSpec$ 
35 THEOREM SILiveSpec  $\Rightarrow SIRun \wedge SIPrioReset$ 
36 └────────────────────────────────────────────────────────────────────────┘

```

Figure 4.13: The *SyncInheritance* module.

TLC finds no error model checking module *SyncInheritance*. None of the specified properties is violated. Intuitively, this seems rather obvious. With such a definition of *SINext*, the system's behavior is not much different from the one specified in module *PInhResourceManagement*. It just takes a few extra steps to update the tasks' active priorities, but the calculus is "immediately after" the event that triggered it. It is confirmed that the delay in the actualization of the task's active priorities is the only source of errors. Even though this could intuitively seem rather obvious, it was worst to verify. Intuitive assumptions

are often a source of errors.

It is now certain that the errors come from the delay between the above mentioned (i–iv) events and the actualization of the tasks' active priorities. But this delay is unavoidable. A distributed system with such a behavior restriction like the one applied in module *SyncInheritance* is not of great usefulness. Let's try to solve the problem in another way.

4.4.2 Divide to Conquer

Module *DistributedInheritance* is a relatively large specification. Consequently, introducing changes in the definition of action *DICalculatePri* and model checking the specification at every change for large instances can take rather long. To speed up this process, we can create a new, considerably smaller, specification. It shall contain simply the “new version” of action *DICalculatePri* and some “strictly necessary” extras. What should these be?

The crux of the problem is to make sure that, after each calculation, every task's active priority is in accordance with the priority inheritance protocol laws. For that, we will make a new specification with only two actions. First, an action (1) that simulates the random generation of the before mentioned (i–iv) events. Secondly, the action (2) that updates the tasks' active priorities. Should action (2) correctly calculate the tasks' active priorities for any random scenario generated by action (1), then it shall do so for the scenarios generated by a distributed specification analogous to module *DistributedInheritance*.

The specification created is module *CalculationScheme* in Figure 4.14. Action *CSFeed* represents action (1), and *CSCalculate* action (2). The definition of invariant *CSPriosCorrect*, at the end of the module, is similar to before.

```

1  ┌────────────────────────── MODULE CalculationScheme ───────────────────┐
2  EXTENDS Naturals, Sequences
3
4  CONSTANTS Task, Resource, NoTask, NoRes, PriMax, QMax
5
6  VARIABLES ResState, TaskOw, TNominalPri, TActivePri, FeedQ
7  CSvars  $\triangleq$   $\langle$  ResState, TaskOw, TNominalPri, TActivePri, FeedQ  $\rangle$ 
8
9  PIInhItem  $\triangleq$  [ Type : { "TLocksR", "TWaitsOnR", "TReleasesR", "TIsNotified" },
10                  OnTask : Task,
11                  OnRes : Resource ]
12 ───────────────────────────────────────────────────────────────────────────┘
13 CSTypeOk  $\triangleq$   $\wedge$  TNominalPri  $\in$  [Task  $\rightarrow$  1 .. PriMax]
14
15                   $\wedge$  ResState  $\in$  [Resource  $\rightarrow$  [Locked : BOOLEAN ,
16                                     WL : SUBSET (Task),
17                                     Owner : Task  $\cup$  { NoTask }]]
18
19                   $\wedge$  TaskOw  $\in$  [Task  $\rightarrow$  [Owning : SUBSET (Resource),
20                                     Pre : Resource  $\cup$  { NoRes }]]
21
22                   $\wedge$  TActivePri  $\in$  [Task  $\rightarrow$  1 .. PriMax]
23
24                   $\wedge$  FeedQ  $\in$  Seq(PIInhItem)

```

```

26  $CSInit \triangleq \wedge TNominalPri \in [Task \rightarrow 1 \dots PriMax]$ 
27    $\wedge ResState = [r \in Resource \mapsto [Locked \mapsto FALSE,$ 
28      $WL \mapsto \{\},$ 
29      $Owner \mapsto NoTask]]$ 
30    $\wedge TaskOw = [t \in Task \mapsto [Owning \mapsto \{\},$ 
31      $Pre \mapsto NoRes]]$ 
32    $\wedge TActivePri = [t \in Task \mapsto TNominalPri[t]]$ 
33    $\wedge FeedQ = \langle \rangle$ 
34 ───────────────────────────────────────────────────────────────────────────────────
35  $CSFeed(item) \triangleq$ 
36    $\wedge Len(FeedQ) < QMax$ 
37    $\wedge LET$ 
38      $type \triangleq item.Type$ 
39      $t \triangleq item.OnTask$ 
40      $r \triangleq item.OnRes$ 
41    $IN$ 
42      $\wedge CASE type = "TLocksR" \rightarrow$ 
43        $\wedge ResState' = [ResState \text{ EXCEPT } ![r].Locked = TRUE,$ 
44          $![r].Owner = t]$ 
45        $\wedge TaskOw' = [TaskOw \text{ EXCEPT } ![t].Owning = @ \cup \{r\}]$ 
46
47      $\square type = "TWaitsOnR" \rightarrow$ 
48        $\wedge ResState' = [ResState \text{ EXCEPT } ![r].WL = @ \cup \{t\}]$ 
49        $\wedge TaskOw' = [TaskOw \text{ EXCEPT } ![t].Pre = r]$ 
50
51      $\square type = "TReleasesR" \rightarrow$ 
52        $\wedge ResState' = [ResState \text{ EXCEPT } ![r].Owner = NoTask,$ 
53          $![r].Locked = FALSE]$ 
54        $\wedge TaskOw' = [TaskOw \text{ EXCEPT } ![t].Owning = @ \setminus \{r\}]$ 
55
56      $\square type = "TIsNotified" \rightarrow$ 
57        $\wedge ResState' = [ResState \text{ EXCEPT } ![r].Locked = TRUE,$ 
58          $![r].Owner = t,$ 
59          $![r].WL = @ \setminus \{t\}]$ 
60        $\wedge TaskOw' = [TaskOw \text{ EXCEPT } ![t].Pre = NoRes,$ 
61          $![t].Owning = @ \cup \{r\}]$ 
62        $\wedge FeedQ' = Append(FeedQ, item)$ 
63        $\wedge UNCHANGED \langle TNominalPri, TActivePri \rangle$ 
64 ───────────────────────────────────────────────────────────────────────────────────
65  $CSCalculate \triangleq$ 
66    $\wedge Len(FeedQ) > 0$ 
67
68    $[ \dots ]$ 
69
70    $\wedge CASE type = "TLocksR" \rightarrow$ 
71      $[ \dots ]$ 
72      $TActivePri' = \dots$ 
73
74    $\square type = "TWaitsOnR" \rightarrow$ 
75      $[ \dots ]$ 
76      $TActivePri' = \dots$ 
77
78    $\square type = "TReleasesR" \rightarrow$ 
79      $[ \dots ]$ 
80      $TActivePri' = \dots$ 
81
82    $\square type = "TIsNotified" \rightarrow$ 
83      $[ \dots ]$ 

```

4.4. DISTRIBUTING THE PRIORITY INHERITANCE COMPUTATIONS 73

```

84      TActivePri' = ...
86      ∧ FeedQ' = Tail(FeedQ)
87      ∧ UNCHANGED ⟨TNominalPri, ResState, TaskOw⟩
88  |-----|
89  CSNext ≜
90      ∨ ∃ item ∈ PInhItem :
91          CSFeed(item)
92      ∨ CSCalculate
94  CSSpec ≜ CSInit ∧ □[CSNext]CSvars
95  |-----|
96  PrioRange ≜ 1 .. PriMax
98  BlockT(t) ≜ {tup ∈ Resource × Task × PrioRange : ∧ tup[1] ∈ TaskOw[t].Owning
99                                     ∧ tup[2] ∈ ResState[tup[1]].WL
100                                    ∧ tup[3] = TActivePri[tup[2]]}
101  |-----|
102  CSPriosCorrect ≜
103      Len(FeedQ) = 0 ⇒
104      ∀ t ∈ Task :
105          IF (BlockT(t) ≠ {})
106          THEN
107              LET
108                  XhgPriTup ≜ CHOOSE tup ∈ BlockT(t) :
109                                     ∀ otup ∈ BlockT(t) : tup[3] ≥ otup[3]
110                  XhgPrio ≜ XhgPriTup[3]
111                  XnewActivePri ≜ IF XhgPrio > TNominalPri[t]
112                                     THEN XhgPrio ELSE TNominalPri[t]
113              IN
114                  TActivePri[t] = XnewActivePri
115          ELSE
116                  TActivePri[t] = TNominalPri[t]
117  |-----|
118  THEOREM CSSpec ⇒ □CSPriosCorrect
119  |-----|

```

Figure 4.14: The *CalculationScheme* module.

Using module *CalculationScheme*, it was possible to experiment many solutions to replace the “...” after each branch of the CASE expression of action *CSCalculate*. None of the attempts worked well. TLC was always able to find behaviors that violated invariant *CSPriosCorrect*.

The error traces were some times rather long. It would have been extremely difficult to foresee them all mentally. This is the great advantage of using a model checking tool. Model checkers simply generate all the possible sequences of events. Any scenario becomes as difficult to “imagine” as any other.

4.4.3 The Conquest: From Sequential to Combinatorial Calculus

At first sight, module *CalculationScheme* was not of great help. Every failure has, however, its lessons. Gradually, each new counter example demonstrated how complex, if not impossible, it would be to develop a solution for calculating the tasks' active priority based on a sequential logic. Indeed, until here, all the attempts were based on the same sequential principle: knowing the current state⁷ and the new event, a new state is calculated. Can we find a different strategy?

The definition of the invariant *CSPriosCorrect* is quite revealing. In fact, it shows that it is possible to, at any time, derive every task's active priority simply based on the current resource allocation map. Actually, in the definition of action *CSCalculate*, we could simply remove the CASE expression and replace it by a block similar to the right hand side of the implication of invariant *CSPriosCorrect*. While effective, such procedure is, however, a "non-solution". After all, the goal is to devise a lower-level implementation.

In the definition of *CSPriosCorrect*, it is used the state function *BlockT(t)*.⁸ *BlockT(t)* returns a set of tuples $\langle ri, ti, pi \rangle$ such that resource ri is owned by task t , task ti is waiting for ri , and pi is the active priority of ti . The priority that task t inherits is the highest priority of the tasks blocked by it, i.e., the highest pi returned by *BlockT(t)*. To reach an implementation of such calculations, let's introduce two new variables, *TOwR* and *RBlT*, where:

$$a) \ TOwR \in [Task \times Resource \rightarrow \text{BOOLEAN}]$$

$$b) \ RBlT \in [Resource \times Task \rightarrow \text{BOOLEAN}]$$

Variable *TOwR* is a function that maps every tuple of task ti and resource ri , $\langle ti, ri \rangle$, to a boolean value. It is defined that *TOwR*[t, r] equals true iff task t owns resource r . On the other hand, variable *RBlT* maps every tuple of resource ri and task ti , $\langle ri, ti \rangle$, to a boolean value. It is defined that *RBlT*[r, t] equals true iff resource r blocks task t (i.e., task t is waiting for resource r).

Let's use an example to better illustrate what is being introduced. Take a system with four tasks $\{t_1, t_2, t_3, t_4\}$ and three resources $\{r_1, r_2, r_3\}$, where tasks try to access resources through the mechanism described in this thesis. Figure 4.15 shows a possible state of such system.

Replacing $\{\text{FALSE}, \text{TRUE}\}$ by $\{0, 1\}$ and writing the result in the form of matrices, for the scenario illustrated in 4.15, variables *TOwR* and *RBlT* will assume the values:

⁷Here, by state we mean the tasks' active priorities.

⁸The state function *BlockT(t)* is first introduced in module *PInhResourceAccess*, in Section 4.2 on page 56.

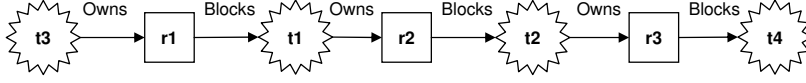


Figure 4.15: Task state diagram.

$$TOwR = \begin{matrix} & r_1 & r_2 & r_3 \\ \begin{matrix} t_1 \\ t_2 \\ t_3 \\ t_4 \end{matrix} & \begin{bmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \\ 1 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix} \end{matrix}, \quad RBIT = \begin{matrix} & t_1 & t_2 & t_3 & t_4 \\ \begin{matrix} r_1 \\ r_2 \\ r_3 \end{matrix} & \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \end{matrix}$$

Observing Figure 4.15 it can be concluded that (i) task t_3 directly blocks task t_1 , which (ii) blocks task t_2 , which in turn (iii) blocks task t_4 . Such conclusion can, however, also be reached mathematically. In fact, if we take $TOwR$ and $RBIT$ as matrices $MTOwR$ and $MRBIT$ and multiply them,⁹ we will have matrix M :

$$M = MTOwR \otimes MRBIT = \begin{bmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \\ 1 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix} \otimes \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

where the (i-iii) blocking situations correspond to the three 1 valued elements of matrix M , namely $M[3,1]$, $M[1,2]$, and $M[2,4]$.

From Figure 4.15 it is also clear that (iv) t_3 indirectly blocks t_2 , (v) t_1 blocks t_4 , and ultimately (vi) t_3 blocks t_4 . Figure 4.16 illustrates the (i-vi) blocking situations.

Again, such conclusions can also be reached mathematically. If we take matrix M and multiply it by itself, we'll have matrix MM :

$$MM = M \otimes M = \begin{bmatrix} 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

where the two 1 valued elements of MM , namely $MM[3,2]$ and $MM[1,4]$, correspond to the "second-order blockings" (iv) and (v).

Adding matrices M and MM ,¹⁰ we obtain matrix N :

⁹Recall that we are dealing with boolean values. Therefore, whenever we refer to matrix multiplication we mean boolean matrix multiplication. We'll use the symbol $\otimes$ to denote the operation.

¹⁰As for multiplication, once we are dealing with boolean values, when we refer to addition we mean logical addition. The symbol $\oplus$ is used to denote the operation.

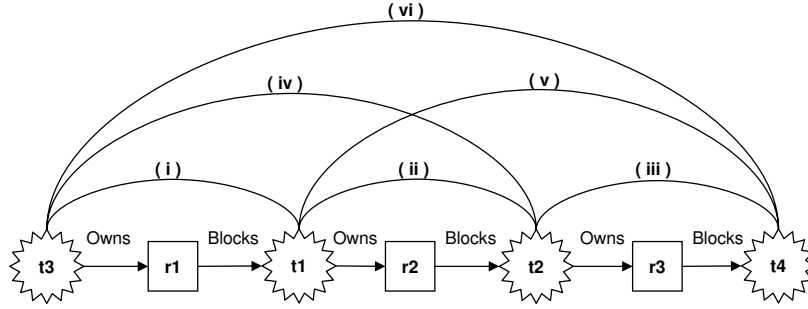


Figure 4.16: Intertask blockings.

$$N = M \uplus MM = \begin{bmatrix} 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 1 \\ 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

Multiplying N by itself, we'll have matrix NN :

$$NN = N \otimes N = \begin{bmatrix} 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

where the new 1 valued element, $NN[3,4]$, corresponds to the “third-order blocking” (vi).

Adding matrices N and NN , we obtain matrix O :

$$O = N \uplus NN = \begin{bmatrix} 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 1 \\ 1 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

where the six 1 valued elements represent the referred (i-vi) blockings.

If we repeat the process, multiplying matrix O by itself, we'll obtain NN again, which means that there are no “higher-order blockings” still to identify.

This process allows us to explicit all the direct and indirect blockings of each task to every other task. The active priority of every task t will be the maximum of t 's nominal priority and the nominal priority of the tasks blocked by t .

The method just presented allows us to calculate every task's active priority at any time directly from an image of the resource allocation map. It is informally summarized in Figure 4.17.

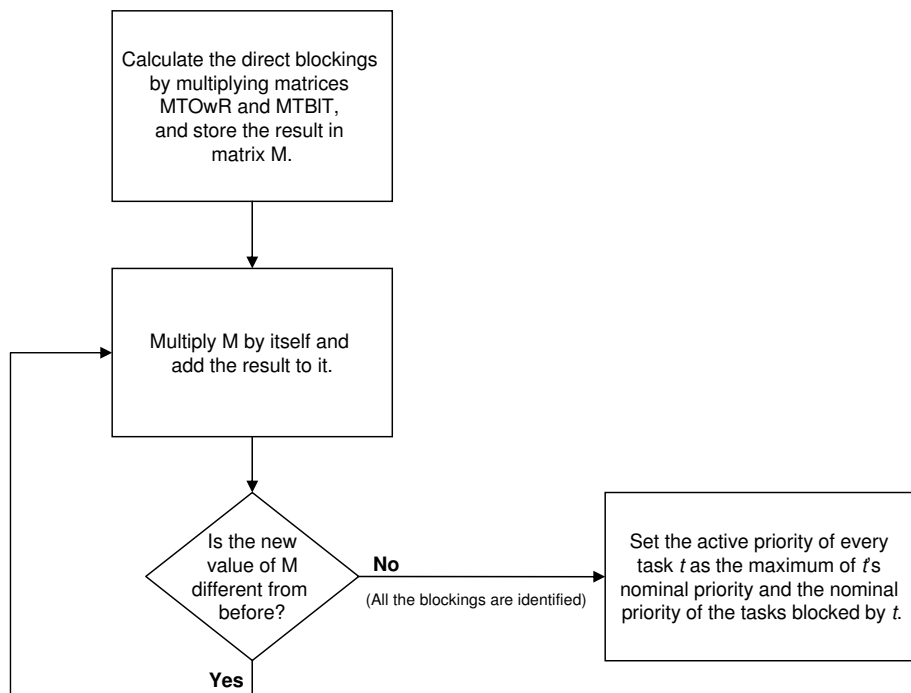


Figure 4.17: Procedure to determine every tasks' active priority.

To formalize the described procedure, a new TLA⁺ module is created. It is called *CombinatorialCalculation* and is shown in Figure 4.18. The module is comparable to module *CalculationScheme* (in Figure 4.14), where the previously discussed action *CSCalculate* is replaced by three new actions, namely *CCCalcA*, *CCCalcB*, and *CCCalcC*. These actions represent the procedure described in Figure 4.17.

The variables *TOwR* and *RBlT*, in the module, have the definition given above. Variable *TBT* represents the matrix *M* of the description in Figure 4.17. The variable *Mlbl* is used to impose the sequential ordering of actions *CCCalcA*, *CCCalcB*, and *CCCalcC*.

Action *CCFeed* is similar to action *CSFeed* from module *CalculationScheme*. It simulates the random generation of (i-iv) events identified in Section 4.4. In each of the branches of the CASE expression of action *CCFeed*, it is updated the value of variables *TOwR* and *RBlT* accordingly to the event in question. Note that variables *ResState* and *TaskOw* are kept simply to allow the expression of the invariants defined in the end of the module.

```

1  ┌────────────────────────── MODULE CombinatorialCalculation ───────────────────────────┐
2  EXTENDS Naturals, Sequences, FiniteSets, TLC
3
4  CONSTANTS Task, Resource, NoTask, NoRes, PriMax
5
6  VARIABLES ResState, TaskOw, TNominalPri, TActivePri, FeedF, RBlT, TOwR, Mlbl, TBT
7  CCvs  $\triangleq$   $\langle \text{ResState}, \text{TaskOw}, \text{TNominalPri}, \text{TActivePri}, \text{FeedF}, \text{RBlT}, \text{TOwR}, \text{Mlbl}, \text{TBT} \rangle$ 
8
9  LOCAL Perms  $\triangleq$  Permutations(Task)  $\cup$  Permutations(Resource)
10
11 LOCAL PInhItem  $\triangleq$  [Type : { "TLocksR", "TWaitOnR", "TReleasesR", "TIsNotified" },
12                      OnTask : Task,
13                      OnRes : Resource]
14 └──────────────────────────────────────────────────────────────────────────────────────────┘
15 CCTypeOk  $\triangleq$ 
16    $\wedge$  TNominalPri  $\in$  [Task  $\rightarrow$  1 .. PriMax]
17
18    $\wedge$  ResState  $\in$  [Resource  $\rightarrow$  [Locked : BOOLEAN ,
19                                     WL : SUBSET (Task),
20                                     Owner : Task  $\cup$  { NoTask }]]
21
22    $\wedge$  TaskOw  $\in$  [Task  $\rightarrow$  [Owning : SUBSET (Resource),
23                                     Pre : Resource  $\cup$  { NoRes }]]
24
25    $\wedge$  TActivePri  $\in$  [Task  $\rightarrow$  1 .. PriMax]
26
27    $\wedge$  FeedF  $\in$  BOOLEAN
28
29    $\wedge$  TOwR  $\in$  [Task  $\times$  Resource  $\rightarrow$  BOOLEAN ]
30
31    $\wedge$  RBlT  $\in$  [Resource  $\times$  Task  $\rightarrow$  BOOLEAN ]
32
33    $\wedge$  Mlbl  $\in$  { "stgA", "stgB", "stgC" }
34
35    $\wedge$  TBT  $\in$  [Task  $\times$  Task  $\rightarrow$  BOOLEAN ]
36
37 CCInit  $\triangleq$ 

```

4.4. DISTRIBUTING THE PRIORITY INHERITANCE COMPUTATIONS 79

```

38  ∧  $TNominalPri \in [Task \rightarrow 1 \dots PriMax]$ 
39  ∧  $ResState = [r \in Resource \mapsto [Locked \mapsto FALSE,$ 
40       $WL \mapsto \{\},$ 
41       $Owner \mapsto NoTask]]$ 
42  ∧  $TaskOw = [t \in Task \mapsto [Owning \mapsto \{\},$ 
43       $Pre \mapsto NoRes]]$ 
44  ∧  $TActivePri = [t \in Task \mapsto TNominalPri[t]]$ 
45  ∧  $FeedF = FALSE$ 
46  ∧  $TOwR = [tup \in Task \times Resource \mapsto FALSE]$ 
47  ∧  $RBIT = [tup \in Resource \times Task \mapsto FALSE]$ 
48  ∧  $Mlbl = \text{"stgA"}$ 
49  ∧  $TBT = [tup \in Task \times Task \mapsto FALSE]$ 
50 |
51   $CCFeed(item) \triangleq$ 
52    ∧  $FeedF = FALSE$ 
53    ∧ LET
54       $type \triangleq item.Type$ 
55       $t \triangleq item.OnTask$ 
56       $r \triangleq item.OnRes$ 
57    IN
58    The two variables "ResState" and "TaskOw" are kept in the model, to be able to check, the
59    way we normally do, whether Priority Inheritance is correctly implemented or not.
60    CASE  $type = \text{"TLocksR"} \rightarrow$ 
61      ∧  $ResState' = [ResState \text{ EXCEPT } ![r].Locked = TRUE,$ 
62           $![r].Owner = t]$ 
63      ∧  $TaskOw' = [TaskOw \text{ EXCEPT } ![t].Owning = @ \cup \{r\}]$ 
64
65      ∧  $TOwR' = [TOwR \text{ EXCEPT } ![t, r] = TRUE]$ 
66      ∧ UNCHANGED  $\langle RBIT \rangle$ 
67
68    □  $type = \text{"TWaitsOnR"} \rightarrow$ 
69      ∧  $ResState' = [ResState \text{ EXCEPT } ![r].WL = @ \cup \{t\}]$ 
70      ∧  $TaskOw' = [TaskOw \text{ EXCEPT } ![t].Pre = r]$ 
71
72      ∧  $RBIT' = [RBIT \text{ EXCEPT } ![r, t] = TRUE]$ 
73      ∧ UNCHANGED  $\langle TOwR \rangle$ 
74
75    □  $type = \text{"TReleasesR"} \rightarrow$ 
76      ∧  $ResState' = [ResState \text{ EXCEPT } ![r].Owner = NoTask,$ 
77           $![r].Locked = FALSE]$ 
78      ∧  $TaskOw' = [TaskOw \text{ EXCEPT } ![t].Owning = @ \setminus \{r\}]$ 
79
80      ∧  $TOwR' = [TOwR \text{ EXCEPT } ![t, r] = FALSE]$ 
81      ∧ UNCHANGED  $\langle RBIT \rangle$ 
82
83    □  $type = \text{"TIsNotified"} \rightarrow$ 
84      ∧  $ResState' = [ResState \text{ EXCEPT } ![r].Locked = TRUE,$ 
85           $![r].Owner = t,$ 
86           $![r].WL = @ \setminus \{t\}]$ 
87      ∧  $TaskOw' = [TaskOw \text{ EXCEPT } ![t].Pre = NoRes,$ 
88           $![t].Owning = @ \cup \{r\}]$ 
89
90      ∧  $TOwR' = [TOwR \text{ EXCEPT } ![t, r] = TRUE]$ 
91      ∧  $RBIT' = [RBIT \text{ EXCEPT } ![r, t] = FALSE]$ 
92
93  ∧  $FeedF' = TRUE$ 
94  ∧ UNCHANGED  $\langle TNominalPri, TActivePri, Mlbl, TBT \rangle$ 
95 |

```

```

96  CCCalcA  $\triangleq$ 
97     $\wedge \text{FeedF} = \text{TRUE}$ 
98     $\wedge \text{Mlbl} = \text{"stgA"}$ 
99     $\wedge \text{LET}$ 
100       $\text{TBTi} \triangleq [tup \in \text{Task} \times \text{Task} \mapsto$ 
101         $\exists r \in \text{Resource} :$ 
102           $\wedge \text{TOwR}[tup[1], r] = \text{TRUE}$ 
103           $\wedge \text{RBLT}[r, tup[2]] = \text{TRUE}]$ 
104      IN
105       $\text{TBTr}' = \text{TBTi}$ 
106     $\wedge \text{Mlbl}' = \text{"stgB"}$ 
107     $\wedge \text{UNCHANGED} \langle \text{ResState}, \text{TaskOw}, \text{TNominalPri}, \text{TActivePri}, \text{FeedF}, \text{RBLT}, \text{TOwR} \rangle$ 

109  CCCalcB  $\triangleq$ 
110     $\wedge \text{FeedF} = \text{TRUE}$ 
111     $\wedge \text{Mlbl} = \text{"stgB"}$ 
112     $\wedge \text{LET}$ 
113       $\text{TBTMult} \triangleq [tup \in \text{Task} \times \text{Task} \mapsto$ 
114         $\exists t \in \text{Task} :$ 
115           $\wedge \text{TBTr}[tup[1], t] = \text{TRUE}$ 
116           $\wedge \text{TBTr}[t, tup[2]] = \text{TRUE}]$ 

118       $\text{MatPlusMult} \triangleq [tup \in \text{Task} \times \text{Task} \mapsto$ 
119         $\vee \text{TBTr}[tup[1], tup[2]] = \text{TRUE}$ 
120         $\vee \text{TBTMult}[tup[1], tup[2]] = \text{TRUE}]$ 
121      IN
122      IF  $\text{TBTr} = \text{MatPlusMult}$ 
123        THEN
124           $\wedge \text{UNCHANGED} \langle \text{TBTr} \rangle$ 
125           $\wedge \text{Mlbl}' = \text{"stgC"}$ 
126        ELSE
127           $\wedge \text{TBTr}' = \text{MatPlusMult}$ 
128           $\wedge \text{UNCHANGED} \langle \text{Mlbl} \rangle$ 
129       $\wedge \text{UNCHANGED} \langle \text{ResState}, \text{TaskOw}, \text{TNominalPri}, \text{TActivePri}, \text{FeedF}, \text{RBLT}, \text{TOwR} \rangle$ 

131  CCCalcC  $\triangleq$ 
132     $\wedge \text{FeedF} = \text{TRUE}$ 
133     $\wedge \text{Mlbl} = \text{"stgC"}$ 
134     $\wedge \text{LET}$ 
135       $\text{pri}(t) \triangleq$ 
136        LET
137           $\text{blockedTasks} \triangleq \{ti \in \text{Task} : \text{TBTr}[t, ti] = \text{TRUE}\}$ 
138        IN
139          IF  $\text{blockedTasks} \neq \{\}$ 
140            THEN
141              LET
142                 $\text{highestPriTask} \triangleq \text{CHOOSE } ti \in \text{blockedTasks} :$ 
143                 $\forall tj \in \text{blockedTasks} : \text{TNominalPri}[ti] \geq \text{TNominalPri}[tj]$ 
144                 $\text{highestPrio} \triangleq \text{TNominalPri}[\text{highestPriTask}]$ 
145                 $\text{newActivePri} \triangleq \text{IF } \text{highestPrio} > \text{TNominalPri}[t]$ 
146                THEN  $\text{highestPrio}$  ELSE  $\text{TNominalPri}[t]$ 
147              IN
148                 $\text{newActivePri}$ 
149            ELSE
150               $\text{TNominalPri}[t]$ 
151          IN
152       $\text{TActivePri}' = [ti \in \text{Task} \mapsto \text{pri}(ti)]$ 
153     $\wedge \text{Mlbl}' = \text{"stgA"}$ 

```

4.4. DISTRIBUTING THE PRIORITY INHERITANCE COMPUTATIONS81

```

544      ∧      FeedF' = FALSE
555      ∧      UNCHANGED ⟨ResState, TaskOw, TNominalPri, RBlT, TOwR, TBT⟩
556 |
557 CCNext ≜
558   ∨   ∃ item ∈ PInhItem :
559       CCFeed(item)
560   ∨   CCCalcA
561   ∨   CCCalcB
562   ∨   CCCalcC
563
564 CCSpec ≜ CCInit ∧ □[CCNext]CCvs
565 |
566 LOCAL PrioRange ≜ 1 .. PriMax
567
568 LOCAL BlockT(t) ≜ { tup ∈ Resource × Task × PrioRange :
569   ∧ tup[1] ∈ TaskOw[t].Owning
570   ∧ tup[2] ∈ ResState[tup[1]].WL
571   ∧ tup[3] = TActivePri[tup[2]] }
572
573 CCActive ≜ ∀ t ∈ Task : TActivePri[t] ≥ TNominalPri[t]
574
575 CCPriosCorrect ≜
576   FeedF = FALSE ⇒
577     ∀ t ∈ Task :
578       (IF (BlockT(t) ≠ {})
579        THEN
580          LET
581            XhgPriTup ≜ CHOOSE tup ∈ BlockT(t) :
582              ∀ otup ∈ BlockT(t) : tup[3] ≥ otup[3]
583            XhgPrio ≜ XhgPriTup[3]
584            XnewActivePri ≜ IF XhgPrio > TNominalPri[t]
585                             THEN XhgPrio ELSE TNominalPri[t]
586          IN
587            TActivePri[t] = XnewActivePri
588        ELSE
589            TActivePri[t] = TNominalPri[t])
590
591 CCBoostedPos ≜
592   FeedF = FALSE ⇒
593     ∀ t ∈ Task :
594       (TActivePri[t] > TNominalPri[t] ⇒ TaskOw[t].Owning ≠ {})
595
596 CCNotBoosted ≜
597   FeedF = FALSE ⇒
598     ∀ t ∈ Task :
599       (TaskOw[t].Owning = {} ⇒ TActivePri[t] = TNominalPri[t])
600
601 CCPInhCalc ≜ CCActive ∧ CCPriosCorrect ∧ CCBoostedPos ∧ CCNotBoosted
602 |
603 THEOREM CCSpec ⇒ □(CCTypeOk ∧ CCPInhCalc)
604 |

```

Figure 4.18: The *CombinatorialCalculation* module.

Model checking module *CombinatorialCalculation*, TLC found no error. The described procedure correctly calculates every task's active priority. The specification was model checked with instances up to four tasks, three resources, and three priority levels (i.e., $PriMax = 3$), or five tasks, two resources, and three priority levels. Model checking such a system takes approximately one day.¹¹

After reaching these results, it was searched out whether similar methods had been published. It so happens that boolean matrices operations have served to implement graph theory algorithms as early as from the 1970s [3, 22]. Nevertheless, the procedure here presented was “discovered” with total unawareness of such fact, which highly emphasizes the power of formal reasoning in the development of computational systems.

4.5 Summary

In the beginning of this chapter, we extended the resource management mechanism developed in Chapter 3 by giving priorities to tasks and granting the access of the tasks to resources based on their priority. Modules *PrioResourceAccess* and *PrioResourceManagement* reflect such changes.

Modules *PInhResourceAccess* and *PInhResourceManagement* were next presented. These modules incorporate the priority inheritance protocol principles, a protocol devised to avoid the possible unbounded blocking time of higher-priority tasks.

Starting from module *PInhResourceManagement*, it was then decided to try to devise a lower level implementation of such specification. For that, it was made the exercise of trying to reach a solution where the priority inheritance computations would be processed independently from the “normal” behavior of the system, namely in a different node. By the end of this exercise, it was reached a solution that allows the calculation of every task's active priority directly from the resource allocation map.

Whether it would be advantageous to implement the devised calculation method in a distributed manner is not obvious. On one hand, the “normal” system could become faster, once it would not need to compute the priority inheritance related code. On the other hand, such a system would rely on the fact that it is acceptable for a task to have its active priority updating delayed. If this is not the case, it would be necessary to impose a restriction similar to the one implemented in module *SyncInheritance*. (But this would considerably lessen the benefits of distributing the system.)

The solution described in module *CombinatorialCalculation* can naturally also be used in a non-distributed implementation. It is a highly conceptually simple method for the implementation of the priority inheritance protocol. Moreover, the method could also be applied to identify the circular wait conditions described in Section 3.3.1. In fact, such a blocking would be equivalent to the occurrence of a symmetric value in the matrix M of Figure 4.17 (or variable TBT in module *CombinatorialCalculation*).

¹¹TLC was run on a single processor 1.8 GHz PC running Windows XP.

It would be straightforward to update all the modules formerly described in this chapter in order to incorporate the last results achieved. For the sake of brevity such work is not presented in this thesis. Figure 4.19 illustrates the modules presented in the chapter.

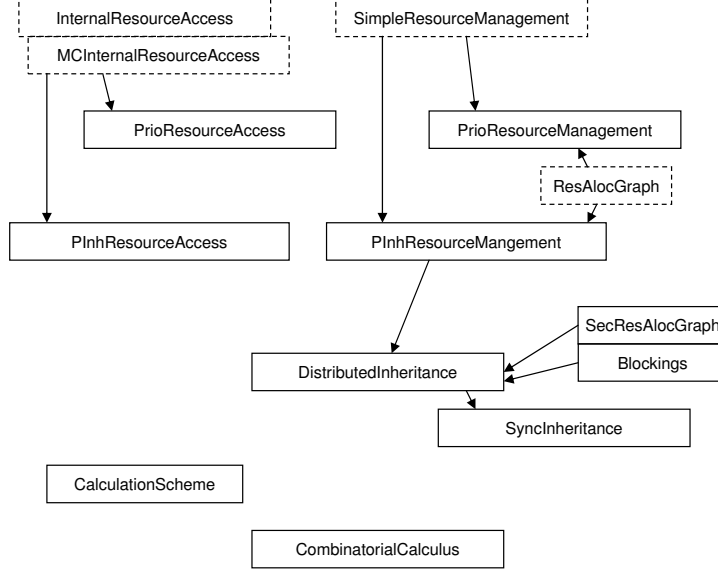


Figure 4.19: Modules presented in this chapter.

Chapter 5

Communication and Synchronization

In this chapter, a mechanism for communication and synchronization between tasks is formally described. The mechanism can be applied to both distributed and non-distributed systems. The major guidelines for this work were introduced in Section 1.2. Arguably, the principles and TLA⁺ descriptions here presented might constitute a basis for the development of “nano-kernel” operating systems. This is addressed at the end of the chapter.

5.1 Intertask communication

The mechanism for communication and synchronization between tasks here developed is close to the ideas presented in Hoare’s seminal work, the CSP language [10, 25]. In CSP, tasks communicate by synchronously exchanging messages through channels. A task outputs a message at the same time as the other inputs it. For asynchronous communication, a virtual buffering task is used.

In our design, tasks communicate by exchanging messages on *ports*. The communication can either be synchronous or asynchronous. In the first case, the task execution is suspended until another “matching” message arrives. In the later, the task may proceed its execution immediately after sending the message away. Ports are also used in, e.g., the Mach micro kernel [2]. The definition is, however, not exactly the same.¹

The messages exchanged are encapsulated into *packets*. A packet contains all the information needed to perform the operation. If the packets carry some data, the tasks are said to *communicate*. When no data is exchanged, the tasks

¹This is a recurring problem in computing science. Adopting the same term for different abstractions can lead to wrong assumptions and be a source of errors. On the other side, a known term might provide an easier understanding of the new abstraction and lead to its better acceptance.

are said to *synchronize*. For short, in the remaining, we will normally refer to communication meaning both situations.

5.1.1 Ports and Tasks Everywhere

Before proceeding, we can *a priori* summarize our solution in a single sentence, our designated “every-every” principle: *everything is a task and every task has a port*. In fact, for each packet that a task t sends to a port, t must receive an answer. This answer is yet a packet that is sent to t by being put into t ’s *entry port*. On the other side, a port p also receives packets in its entry port, and p ’s activity is performed by its own “virtual task”.

To fit this mechanism in the context of an operating system, it is necessary to introduce another element, the *kernel*. The kernel is the central processing entity. It coordinates all the system activity. Its behavior is, however, very simple. It merely delivers packets to proper destinations, either ports or tasks. Better said, it only delivers packets to entry ports, whether these are ports’ entry ports or tasks’ entry ports. The kernel also fits into the “every-every” principle. Indeed, packets are sent to the kernel by being put into its *entry port*, and the kernel is just a “special task”.

Every communication is executed through the kernel. In fact, tasks do not directly send packets to specific destination ports. They always send the packets to the kernel, which then forwards them. Like that, tasks can be considered “blind”.

5.1.2 The TLA⁺ Specification

The developed intertask communication mechanism is depicted in Figure 5.1. It illustrates a configuration with two tasks, $t1$ and $t2$, a port $p1$, and the kernel. Brief and informally, the different kinds of events are: (i) tasks $t1$ and $t2$ send packets to the kernel entry port, which (ii) forwards the packets to port $p1$ entry port. When the conditions for synchronization are met, (iii) port $p1$ returns the two packets to the kernel entry port. The kernel then (iv) forwards the packets to each task entry port. Finally, (v) each task assimilates the information contained in the received packet and releases it. Because the kernel is only involved in (ii) and (iv), its definition is extraordinarily simple and concise.

The TLA⁺ specification is module *IntertaskCommunication* presented on the next pages. It starts with the CONSTANTS statement. Parameters *Task* and *Port* represent the set of tasks and the set of ports, respectively. The *MemAdr* parameter introduces a new element not referred yet. In fact, we have imprecise in the description of the (i–v) events above. In the specification, a packet is mapped to a memory location and what “circulates” among the different entities is a reference to the packet. The parameter *MemAdr* represents the set of memory addresses.

Variable *Mem* represents the mapping of each memory address to a packet, or to *NoPac*. An assignment of a memory address ma to *NoPac* represents that

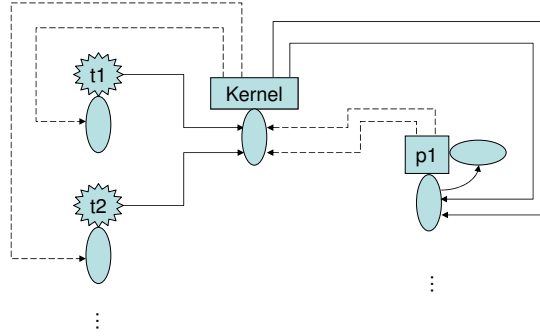


Figure 5.1: The intertask communication mechanism.

ma is free. The structure of a packet is defined by the record *Packet*:

$$\begin{aligned}
 ICServ &\triangleq \{ \text{"orig"}, \text{"comp"}, \text{"acknow"} \} \\
 ICServType &\triangleq \{ \text{"sync"}, \text{"async"} \} \\
 \text{LOCAL } Packet &\triangleq [src : Task, \\
 &\quad dest : Port, \\
 &\quad serv : ICServ, \\
 &\quad stype : ICServType]
 \end{aligned}$$

A *Packet* is a record with four fields. Field *src* identifies the task that originated the request. Field *dest* is the port where the task wants to communicate/synchronize. For two packets to synchronize in a port, it is necessary that they represent complementary requests. To identify the kind of the request it is used field *serv*. It can have three different values, “orig”, “comp”, and “acknow”. The two first represent the two kinds of requests that a task may generate. The later is written in the packet after synchronization in a port.² A packet with “acknow” written in the *serv* field is a packet “returning” to a task. At last, field *stype* is used to distinguish whether the request is synchronous or asynchronous.

In an implementation, the packets should also carry the data to be exchanged between the two tasks. This could be done by simply adding a *data* field to the definition of *Packet*. However, at this level of abstraction, such aspect was not considered significant and, therefore, not represented.

When a task t issues a synchronous request, its activity is suspended until the reception of an answer. To represent this, it is used variable $TReqOk$ that maps each task to a boolean value. If $TReqOk[t]$ equals false that means that task t is waiting for the answer to arrive. When that happens, $TReqOk[t]$ is reset to true. In asynchronous requests, $TReqOk[t]$ is left unchanged (it remains true).

The other three variables, $KerEP$, $TaskEP$, and $PortEP$ represent the entry ports of the kernel, every task, and every port, respectively. Variable $PortEP$ is a record with two fields, *Entry* and *Store*. The distinction is due to the following:

²Values “orig” and “comp” could be replaced by several services, like, e.g., “read” and “write”, “send” and “receive”, “synca” and “syncb”, etc..

when a packet, or, more precisely, a reference ma to the packet is sent by the kernel to a port p , ma is simply added to $PortEP[p].Entry$. The packet is then “analyzed” in the context of the port. If there is a complementary request stored in the port, i.e., in $PortEP[p].Store$, the two packets “synchronize”. Otherwise, ma is moved to $PortEP[p].Store$. This is represented in action $ICPortOp$.

Action $ICIssueRequest$ represents the issuing of a communication request by a task t . For that, it is necessary that there is a free memory address ma , i.e., that $Mem[ma]=NoPac$ and that $TReqOk[t]$ equals true. The identification of the requesting task t , destination port p , kind of request, either “orig” or “comp”, and type of request, either “sync” or “async”, is registered in a packet pac that is mapped to ma in Mem . Finally, the value ma is added to $KerEP$ and $TReqOk$ is updated depending on whether the request is synchronous or asynchronous.

Action $ICKernelOp$ represents the activity of the kernel. It merely moves the memory addresses in its entry port to a proper destination. For each memory address ma , the kernel checks which is the service s of the packet pac linked to ma . If s equals “acknow”, ma is sent to the task indicated in the src field of pac . Otherwise, ma is sent to the port registered in the packet’s $dest$ field.

The activity in the ports is represented by action $ICPortOp$ as described above. All the existential quantifications are left outside the definition of the action due to the reasons explained in Section 3.5.4 (page 45).

Action $ICTaskOp$ represents the operation of a task t when it receives the answer to its request. The memory address linked to the packet is set free and $TReqOk[t]$ is reset to true had it been a synchronous request.

The next-state relation $ICNext$ is the disjunction of all the actions. At the bottom, several safety and liveness properties are specified. Their definition is easily readable.

Note that the all entry ports were defined as sets. In a real implementation, they would probably be implemented as some form of list, either prioritized or FIFO based. Here, no restriction of such kind is made. Like this, the specification does not constrain the behavior *a priori* and is valid for any kind of implementation.

1 MODULE *IntertaskCommunication*

2 CONSTANTS $MemAdr, Task, Port, NoPac$

4 VARIABLES $Mem, KerEP, PortEP, TReqOk, TaskEP$

5 $ICvars \triangleq \langle Mem, KerEP, PortEP, TReqOk, TaskEP \rangle$

7 $ICService \triangleq \{ \text{“orig”, “comp”, “acknow”} \}$

8 $ICServType \triangleq \{ \text{“sync”, “async”} \}$

9 LOCAL $Packet \triangleq [src : Task,$

10 $dest : Port,$

11 $serv : ICService,$

12 $stype : ICServType]$

14 $ICTypeOk \triangleq$

15 $\wedge Mem \in [MemAdr \rightarrow Packet \cup \{NoPac\}]$

```

16   $\wedge TReqOk \in [Task \rightarrow \text{BOOLEAN}]$ 
17   $\wedge KerEP \in \text{SUBSET}(MemAdr)$ 
18   $\wedge TaskEP \in [Task \rightarrow \text{SUBSET}(MemAdr)]$ 
19   $\wedge PortEP \in [Port \rightarrow [Entry : \text{SUBSET}(MemAdr),$ 
20     $Store : \text{SUBSET}(MemAdr)]]$ 

22   $ICInit \triangleq$ 
23     $\wedge Mem = [ma \in MemAdr \mapsto NoPac]$ 
24     $\wedge TReqOk = [t \in Task \mapsto \text{TRUE}]$ 
25     $\wedge KerEP = \{\}$ 
26     $\wedge TaskEP = [t \in Task \mapsto \{\}]$ 
27     $\wedge PortEP = [p \in Port \mapsto [Entry \mapsto \{\},$ 
28       $Store \mapsto \{\}]]$ 

29  |-----|
30   $ICIssueRequest(ma, t, p, s, st) \triangleq$ 
31     $\wedge TReqOk[t] = \text{TRUE}$ 
32     $\wedge Mem[ma] = NoPac$ 
33     $\wedge \text{LET}$ 
34       $pac \triangleq \text{CHOOSE } paci \in Packet : \wedge paci.src = t$ 
35       $\wedge paci.dest = p$ 
36       $\wedge paci.serv = s$ 
37       $\wedge paci.stype = st$ 
38     $\text{IN}$ 
39       $Mem' = [Mem \text{ EXCEPT } ![ma] = pac]$ 
40       $\wedge KerEP' = KerEP \cup \{ma\}$ 
41       $\wedge TReqOk' = \text{IF } st = \text{"sync"} \text{ THEN } [TReqOk \text{ EXCEPT } ![t] = \text{FALSE}] \text{ ELSE } TReqOk$ 
42       $\wedge \text{UNCHANGED } \langle PortEP, TaskEP \rangle$ 

44   $ICKernelOp(ma) \triangleq$ 
45     $\wedge ma \in KerEP$ 
46     $\wedge \text{LET}$ 
47       $pac \triangleq Mem[ma]$ 
48       $p \triangleq pac.dest$ 
49       $t \triangleq pac.src$ 
50       $s \triangleq pac.serv$ 
51     $\text{IN}$ 
52       $\text{IF } s \neq \text{"acknow"}$ 
53       $\text{THEN}$ 
54         $\wedge PortEP' = [PortEP \text{ EXCEPT } ![p].Entry = @ \cup \{ma\}]$ 
55         $\wedge \text{UNCHANGED } \langle TaskEP \rangle$ 
56       $\text{ELSE}$ 
57         $\wedge TaskEP' = [TaskEP \text{ EXCEPT } ![t] = @ \cup \{ma\}]$ 
58         $\wedge \text{UNCHANGED } \langle PortEP \rangle$ 
59       $\wedge KerEP' = KerEP \setminus \{ma\}$ 
60       $\wedge \text{UNCHANGED } \langle TReqOk, Mem \rangle$ 

62   $ICPortOp(p, ma, xma) \triangleq$ 
63     $\wedge ma \in PortEP[p].Entry$ 
64     $\wedge \text{LET}$ 
65       $pac \triangleq Mem[ma]$ 
66       $s \triangleq pac.serv$ 
67       $StoEP \triangleq PortEP[p].Store$ 
68     $\text{IN}$ 
69       $\text{IF } ((StoEP = \{\}) \vee (\forall mai \in StoEP : Mem[mai].serv = s))$ 
70       $\text{THEN}$  If  $DepQ$  is empty or there is no matching request, store the request
71       $\wedge PortEP' = [PortEP \text{ EXCEPT } ![p].Store = @ \cup \{ma\},$ 
72       $![p].Entry = @ \setminus \{ma\}]$ 
73       $\wedge \text{UNCHANGED } \langle Mem, KerEP \rangle$ 

```

```

74      ELSE If there is a complementary request, "synchronize"
75           $\wedge xma \in StoEP$ 
76           $\wedge PortEP' = [PortEP \text{ EXCEPT } ![p].Store = @ \setminus \{xma\},$ 
77                       $![p].Entry = @ \setminus \{ma\}]$ 
78           $\wedge Mem' = [Mem \text{ EXCEPT } ![ma].serv = \text{"acknow"},$ 
79                       $![xma].serv = \text{"acknow"}]$ 
80           $\wedge KerEP' = KerEP \cup \{ma, xma\}$ 
81       $\wedge \text{UNCHANGED } \langle TReqOk, TaskEP \rangle$ 

83   $ICTaskOp(t, ma) \triangleq$ 
84       $\wedge ma \in TaskEP[t]$ 
85       $\wedge \text{LET}$ 
86           $pac \triangleq Mem[ma]$ 
87           $st \triangleq pac.stype$ 
88       $\text{IN}$ 
89           $TReqOk' = \text{IF } st = \text{"sync"} \text{ THEN } [TReqOk \text{ EXCEPT } ![t] = \text{TRUE}] \text{ ELSE } TReqOk$ 
90           $\wedge Mem' = [Mem \text{ EXCEPT } ![ma] = NoPac]$ 
91           $\wedge TaskEP' = [TaskEP \text{ EXCEPT } ![t] = @ \setminus \{ma\}]$ 
92           $\wedge \text{UNCHANGED } \langle KerEP, PortEP \rangle$ 
93  ────────────────────────────────────────────────────────────────────────────────────────────────────
94   $ICNext \triangleq$ 
95       $\exists ma \in MemAdr :$ 
96           $\vee \exists t \in Task :$ 
97               $\exists p \in Port :$ 
98                   $\exists s \in ICServ \setminus \{\text{"acknow"}\} :$ 
99                       $\exists st \in ICServType :$ 
100                           $ICIssueRequest(ma, t, p, s, st)$ 
101           $\vee ICKernelOp(ma)$ 
102           $\vee \exists t \in Task :$ 
103               $ICTaskOp(t, ma)$ 
104           $\vee \exists p \in Port :$ 
105               $\exists xma \in MemAdr :$ 
106                   $ICPortOp(p, ma, xma)$ 

108   $ICSpec \triangleq ICAInit \wedge \Box [ICNext]_{ICvars}$ 
109  ────────────────────────────────────────────────────────────────────────────────────────────────────
110   $ICUniqueSet \triangleq$  An address can only be in one EP at a time
111       $\forall t \in Task :$ 
112           $\forall p \in Port :$ 
113               $\forall si \in \{TaskEP[t], PortEP[p].Entry, PortEP[p].Store, KerEP\} :$ 
114                   $\forall sj \in \{TaskEP[t], PortEP[p].Entry, PortEP[p].Store, KerEP\} \setminus \{si\} :$ 
115                       $si \cap sj = \{\}$ 

117   $ICMemClean \triangleq$  A clean is free iff it does not belong do any EP
118       $\forall ma \in MemAdr : Mem[ma] \neq NoPac \equiv$ 
119           $\vee ma \in KerEP$ 
120           $\vee \exists p \in Port :$ 
121               $\vee ma \in PortEP[p].Entry$ 
122               $\vee ma \in PortEP[p].Store$ 
123           $\vee \exists t \in Task :$ 
124               $ma \in TaskEP[t]$ 

126   $ICTaskEPOk \triangleq$ 
127       $\forall t \in Task :$ 
128           $\forall ma \in TaskEP[t] :$ 
129               $\wedge Mem[ma].src = t$ 
130               $\wedge Mem[ma].serv = \text{"acknow"}$ 

```

```

132 ICPortEPOk  $\triangleq$ 
133    $\forall p \in Port :$ 
134      $\wedge \forall mai \in PortEP[p].Entry :$ 
135        $\wedge Mem[mai].dest = p$ 
136        $\wedge Mem[mai].serv \neq \text{"acknow"}$ 
137      $\wedge \forall maj \in PortEP[p].Store :$ 
138        $\wedge Mem[maj].dest = p$ 
139        $\wedge Mem[maj].serv \neq \text{"acknow"}$ 

141 ICUniformStore  $\triangleq$ 
142    $\forall p \in Port :$ 
143      $\forall mai, maj \in PortEP[p].Store :$ 
144        $Mem[mai].serv = Mem[maj].serv$ 

146 ICSafe  $\triangleq$  ICUniqueSet  $\wedge$  ICMemClean  $\wedge$  ICTaskEPOk  $\wedge$  ICPortEPOk  $\wedge$  ICUniformStore

148 ICMoving  $\triangleq$ 
149    $\wedge \forall ma \in MemAdr : \Box \Diamond (Mem[ma] = NoPac)$ 
150    $\wedge \forall t \in Task : \Box \Diamond (TReqOk[t] = TRUE)$ 
151
```

Figure 5.2: The *IntertaskCommunication* module.

5.1.3 Constraints for Model Checking

Model checking module *IntertaskCommunication* leads to the detection of a problem: the system deadlocks. The problem is that, in the specification, nothing forces tasks to issue requests that complement each other. Therefore, it is perfectly possible to reach a state where every task has issued some synchronous request that does not complement any of the requests issued by the other tasks and, consequently, the system deadlocks.

To avoid deadlocks caused by the scenario described, we will constrain the specification. For that, a new module *MCIntertaskCommunication* is created. It extends module *IntertaskCommunication* and its next-state relation *MCICNext* is similar to *ICNext*, except that action *ICIssueRequest* is substituted by action *MCICIssueRequest*.

Action *MCICIssueRequest* is defined as the conjunction of action *ICIssueRequest* and a conjunct that disables every step that leads to deadlock. Intuitively, it works as follows: when there are more than one task than can issues requests and more than one memory location free, no restriction is imposed. If that is not the case—which means that there might be no possibility of issuing any other request after this one—it is imposed that the request being issued is complementary to an existing one for which there was still no complementary request created.

Module *MCInternalCommunication* is shown in Figure 5.3. The weak fairness conjuncts in the liveness expression *MCICLive* asserts that every issued

request is “processed”. In order to satisfy property *ICMoving* it is also necessary to specify strong fairness of action *MCICIssueRequest*.³ Intuitively, this is because it is necessary to keep issuing new requests to make sure that for each request created eventually there will be a complementary one.

5.2 Distributing the System

The specification in the previous section defines a mechanism for intertask communication. We will now extend it for distributed systems.

In the abstraction, every possible physical computation device is designated a *node*. Tasks and ports are spread over the different nodes. A task located on a certain node shall be able to communicate with other tasks independently from the location of these and of the port used for communication. For instance, a task *ti*, located on node *na*, may communicate with a task *tj*, located on node *nb*, through a port *p*, located on node *nc*.

5.2.1 The External Communication Agents

To allow the communication between tasks located on different nodes, it is necessary to build a mechanism for message passing between the nodes. For that, we will add two new entities to our system, namely, the *sender* and the *receiver*. In practice, the sender and the receiver are “special tasks” that also possess an entry port each other. Each node shall have a sender and a receiver.

The behavior of both the sender and the receiver is quite simple. Whenever a request that arrives to the kernel has as destination an entity that is not located on the present node, the kernel forwards the request to the sender entry port. The sender will then be responsible for exporting the packet containing the request. On the other side, whenever an external packet arrives at a node, it is the receiver that collects it. The receiver has a “blind” activity, it forwards every request to the kernel.

To formalize this specification, module *ExternalPacketFlow* was built. The module is shown next, in Figure 5.5, and abstracts away many of the aspects previously taken in consideration in the specification of module *IntertaskCommunication* (Figure 5.2, page 91).

In module *ExternalPacketFlow*, a packet is defined as a record of two fields:

$$\begin{aligned} \text{LOCAL Location} &\triangleq \{\text{“local”}, \text{“remote”}\} \\ \text{LOCAL Packet} &\triangleq [\text{srcloc} : \text{Location}, \\ &\quad \text{destloc} : \text{Location}] \end{aligned}$$

In fact, for this abstraction, it is not significant what the request is. A packet is simply seen as an entity with a source and a destination, whose location can either be “local” or “remote”. Module *ExternalPacketFlow* models the behavior in one node. A “local” location means that the entity is on the modeled node; a “remote” location means that the entity is located on any other node.

³The difference between weak and strong fairness has been introduced in Section 3.3.

```

1  ┌────────────────── MODULE MCIntertaskCommunication ───────────────────┐
2  EXTENDS IntertaskCommunication, Naturals, FiniteSets

4  LOCAL RdTS  $\triangleq$   $\{ti \in Task : TReqOk[ti] = TRUE\}$ 
5  LOCAL FreeMem  $\triangleq$   $\{mai \in MemAdr : Mem[mai] = NoPac\}$ 

7  MCICIssueRequest(ma, t, p, s, st)  $\triangleq$ 
8     $\wedge ICIssueRequest(ma, t, p, s, st)$ 
9     $\wedge \vee$  Either there is still “place” for more than one request
10       $((Cardinality(RdTS) > 1) \wedge (Cardinality(FreeMem) > 1))$ 
11       $\vee$  Or the new request must be complementary to an existent one that still had
12        no complementary request created
13         $\wedge \exists mai \in MemAdr : \wedge Mem[mai] \neq NoPac$ 
14           $\wedge Mem[mai].dest = p$ 
15           $\wedge Mem[mai].serv \neq s$ 
16         $\wedge \neg \exists maj \in MemAdr : \wedge Mem[maj] \neq NoPac$ 
17           $\wedge Mem[maj].dest = p$ 
18           $\wedge Mem[maj].serv = s$ 
19 └──────────────────────────────────────────────────────────────────────────┘

20 MCICNext  $\triangleq$ 
21    $\exists ma \in MemAdr :$ 
22      $\vee \exists t \in Task :$ 
23        $\exists p \in Port :$ 
24          $\exists s \in ICServ \setminus \{\text{“acknow”}\} :$ 
25          $\exists st \in ICServType :$ 
26            $MCICIssueRequest(ma, t, p, s, st) *$ 
27      $\vee ICKernelOp(ma)$ 
28      $\vee \exists p \in Port :$ 
29        $\exists xma \in MemAdr :$ 
30          $ICPortOp(p, ma, xma)$ 
31      $\vee \exists t \in Task :$ 
32        $ICTaskOp(t, ma)$ 

34 MCICLive  $\triangleq$ 
35    $\forall ma \in MemAdr :$ 
36      $\wedge \forall t \in Task :$ 
37        $\forall p \in Port :$ 
38          $\forall s \in ICServ \setminus \{\text{“acknow”}\} :$ 
39          $\forall st \in ICServType :$ 
40            $SF_{ICvars}(MCICIssueRequest(ma, t, p, s, st))$ 
41      $\wedge WF_{ICvars}(ICKernelOp(ma))$ 
42      $\wedge \forall t \in Task :$ 
43        $WF_{ICvars}(ICTaskOp(t, ma))$ 
44      $\wedge \forall p \in Port :$ 
45        $\forall xma \in MemAdr :$ 
46          $WF_{ICvars}(ICPortOp(p, ma, xma))$ 

48 MCICSpec  $\triangleq ICInit \wedge \Box[MCICNext]_{ICvars}$ 
49 MCICLiveSpec  $\triangleq MCICSpec \wedge MCICLive$ 
50 ┌──────────────────────────────────────────────────────────────────────────┐
51 THEOREM MCICSpec  $\Rightarrow \Box(ICTypeOk \wedge ICSafe)$ 
52 THEOREM MCICLiveSpec  $\Rightarrow ICMoving$ 
53 └──────────────────────────────────────────────────────────────────────────┘

```

Figure 5.3: The *MCIntertaskCommunication* module.

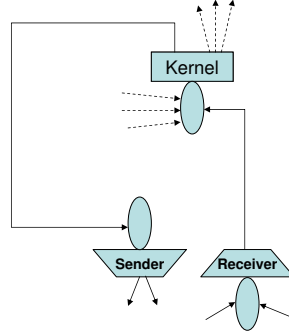


Figure 5.4: The external communication agents.

Variables *Mem* and *KerEP* have the same meaning as before (in module *IntertaskCommunication*). Variables *RcvEP* and *SndEP* represent the receiver's entry port and the sender's entry port, respectively. Initially, all the entry ports are empty and the memory is clean.

Action *EPFPacketArrives* represents the arrival of a packet at the node. The packet is stored in the memory and the memory address filled *ma* is added to the receiver's entry port. The packet has remote source, and the destination can either be local or remote. Every request collected by the receiver is forward to the kernel. This is represented by action *EPFRcvSendsToKernel*.

There can also be requests with local source circulating on the node. The generation of these requests is represented by action *EPFIssueInternalRequest*. In this abstraction, it is not relevant which task originated the request. Tasks are not represented. The local packet is stored in the memory and the memory address filled *ma* is added to *KerEP*.

Action *EPFKernelOp* represents the activity of the kernel. The requests in the kernel entry port are processed as follows: if they have remote destination the kernel forwards them to the sender's entry port. Otherwise, they are simply removed from the kernel's entry port. This is because any request having a local destination is to be either sent to a port or a task. Once neither tasks nor ports are represented, the kernel simply removes the requests from *KerEP*.

Every request that arrives at the sender is exported. In our abstraction this means that the memory element associated with the packet is cleaned and removed from *SndEP*. Hence, the request simply disappears. This is represented by action *EPFSndForwardsPacket*. It perfectly fits our needs. The goal is to represent the activity in one node; the connection to other nodes is abstracted away.

Action *EPFMemoryIsSetFree* represents the cleaning of the memory elements that no longer belong to any entry port. These are the elements that had local destination and that the kernel had simply removed from its entry port in action *EPFKernelOp*.

The next-state relation *EPFNext* is the disjunction of all the actions. The

liveness condition *EPFLive* asserts that every issued request is eventually processed. The properties specified at the end are straightforward.

```

1  ┌────────────────────────────────── MODULE ExternalPacketFlow ───────────────────┐
2  CONSTANTS MemAdr, NoPac

4  VARIABLES Mem, RcvEP, SndEP, KerEP
5  EPFvars  $\triangleq$   $\langle Mem, RcvEP, SndEP, KerEP \rangle$ 

7  LOCAL Location  $\triangleq$  { "local", "remote" }
8  LOCAL Packet  $\triangleq$  [srcloc : Location,
9                    destloc : Location]

11 EPFTypeOk  $\triangleq$ 
12    $\wedge Mem \in [MemAdr \rightarrow Packet \cup \{NoPac\}]$ 
13    $\wedge RcvEP \in \text{SUBSET } (MemAdr)$ 
14    $\wedge SndEP \in \text{SUBSET } (MemAdr)$ 
15    $\wedge KerEP \in \text{SUBSET } (MemAdr)$ 

17 EPFInit  $\triangleq$ 
18    $\wedge Mem = [ma \in MemAdr \mapsto NoPac]$ 
19    $\wedge RcvEP = \{\}$ 
20    $\wedge SndEP = \{\}$ 
21    $\wedge KerEP = \{\}$ 
22 └──────────────────────────────────────────────────────────────────────────────────┘

23 EPFPacketArrives(ma, loc)  $\triangleq$ 
24    $\wedge Mem[ma] = NoPac$ 
25    $\wedge \text{LET}$ 
26      $pac \triangleq \text{CHOOSE } paci \in Packet : \wedge paci.srcloc = \text{"remote"}$ 
27      $\wedge paci.destloc = loc$ 
28    $\text{IN}$ 
29      $Mem' = [Mem \text{ EXCEPT } ![ma] = pac]$ 
30      $\wedge RcvEP' = RcvEP \cup \{ma\}$ 
31      $\wedge \text{UNCHANGED } \langle SndEP, KerEP \rangle$ 

33 EPFIssueInternalRequest(ma, loc)  $\triangleq$ 
34    $\wedge Mem[ma] = NoPac$ 
35    $\wedge \text{LET}$ 
36      $pac \triangleq \text{CHOOSE } paci \in Packet : \wedge paci.srcloc = \text{"local"}$ 
37      $\wedge paci.destloc = loc$ 
38    $\text{IN}$ 
39      $Mem' = [Mem \text{ EXCEPT } ![ma] = pac]$ 
40      $\wedge KerEP' = KerEP \cup \{ma\}$ 
41      $\wedge \text{UNCHANGED } \langle RcvEP, SndEP \rangle$ 

43 EPFRcvSendsToKernel(ma)  $\triangleq$ 
44    $\wedge ma \in RcvEP$ 
45    $\wedge KerEP' = KerEP \cup \{ma\}$ 
46    $\wedge RcvEP' = RcvEP \setminus \{ma\}$ 
47    $\wedge \text{UNCHANGED } \langle SndEP, Mem \rangle$ 

49 EPFSndForwardsPacket(ma)  $\triangleq$ 
50    $\wedge ma \in SndEP$ 
51    $\wedge Mem' = [Mem \text{ EXCEPT } ![ma] = NoPac]$ 
52    $\wedge SndEP' = SndEP \setminus \{ma\}$ 
53    $\wedge \text{UNCHANGED } \langle RcvEP, KerEP \rangle$ 

```

```

55  $EPFKernelOp(ma) \triangleq$ 
56    $\wedge ma \in KerEP$ 
57    $\wedge LET$ 
58      $pac \triangleq Mem[ma]$ 
59    $IN$ 
60      $IF pac.destloc = "local"$ 
61        $THEN$  local destination
62          $UNCHANGED \langle SndEP \rangle$ 
63        $ELSE$  remote destination
64          $SndEP' = SndEP \cup \{ma\}$ 
65      $\wedge KerEP' = KerEP \setminus \{ma\}$ 
66      $\wedge UNCHANGED \langle RcvEP, Mem \rangle$ 

68  $EPFMemoryIsSetFree(ma) \triangleq$ 
69    $\wedge Mem[ma] \neq NoPac$ 
70    $\wedge \forall s \in \{KerEP, SndEP, RcvEP\} : ma \notin s$ 
71    $\wedge Mem' = [Mem \text{ EXCEPT } ![ma] = NoPac]$ 
72    $\wedge UNCHANGED \langle KerEP, SndEP, RcvEP \rangle$ 
73 |-----|
74  $EPFNext \triangleq$ 
75    $\exists ma \in MemAdr :$ 
76      $\vee \exists loc \in Location :$ 
77        $\vee EPFPacketArrives(ma, loc)$ 
78        $\vee EPFIssueInternalRequest(ma, loc)$ 
79        $\vee EPFRcvSendsToKernel(ma)$ 
80        $\vee EPFSndForwardsPacket(ma)$ 
81        $\vee EPFKernelOp(ma)$ 
82        $\vee EPFMemoryIsSetFree(ma)$ 

84  $EPFLive \triangleq$ 
85    $\forall ma \in MemAdr :$ 
86      $\wedge WF_{EPFvars}(EPFRcvSendsToKernel(ma))$ 
87      $\wedge WF_{EPFvars}(EPFSndForwardsPacket(ma))$ 
88      $\wedge WF_{EPFvars}(EPFKernelOp(ma))$ 
89      $\wedge WF_{EPFvars}(EPFMemoryIsSetFree(ma))$ 

91  $EPFSpec \triangleq EPFInit \wedge \Box[EPFNext]_{EPFvars}$ 
92  $EPFLiveSpec \triangleq EPFSpec \wedge EPFLive$ 
93 |-----|
94  $EPFUniqueSet \triangleq$  An address can only be in one EP at a time
95    $\forall si \in \{KerEP, SndEP, RcvEP\} :$ 
96      $\forall sj \in \{KerEP, SndEP, RcvEP\} \setminus \{si\} :$ 
97        $si \cap sj = \{\}$ 

99  $EPFMemNotBlanck \triangleq$  An address in use is not blanck
100    $\forall ma \in MemAdr :$ 
101      $(\exists s \in \{KerEP, SndEP, RcvEP\} : ma \in s) \Rightarrow (Mem[ma] \neq NoPac)$ 

103  $EPFSndEPOk \triangleq$  All packets in SndEP have remote destination
104    $\forall ma \in SndEP : Mem[ma].destloc = "remote"$ 

106  $EPFRcvEPOk \triangleq$  All packets in RcvEP have remote source
107    $\forall ma \in RcvEP : Mem[ma].srcloc = "remote"$ 

109  $EPFSafe \triangleq EPFUniqueSet \wedge EPFMemNotBlanck \wedge EPFSndEPOk \wedge EPFRcvEPOk$ 

111  $EPFMoving \triangleq$  Every address is infinitely often free
112    $\forall ma \in MemAdr : \Box \Diamond (Mem[ma] = NoPac)$ 

```

113	
114	THEOREM $EPFSpec \Rightarrow \Box(EPFTypeOk \wedge EPFSafe)$
115	THEOREM $EPFLiveSpec \Rightarrow EPFMoving$
116	

Figure 5.5: The *ExternalPacketFlow* module.

Model checking the specification, TLC finds no error. Note that this time it is not necessary to add any extra constraints to the specification to model check it. It is not necessary to “force” any kind of behavior. The size of the state space explored is intrinsically limited by the number of elements of *MemAdr*.

The attentive reader may easily find a possible “optimization” to the specification in module *ExternalPacketFlow*. Indeed, it could be so that the requests arriving at a node but whose destination is yet another node are directly send by the receiver to the sender, short-circuiting the passage through the kernel. However, for that, the receiver would need to check the destination of every packet. Its activity would no longer be “blind”. It is not in the scope of this thesis to discuss further the advantages and disadvantages of such solution compared to the one presented.

5.3 Distributed Communication

The elements introduced in the specifications of modules *IntertaskCommunication* and *ExternalPacketFlow* can be put together to build a distributed communication specification. Figure 5.6 represents a node combining all the entities previously mentioned. It illustrates a configuration with two tasks, one port, and six memory addresses.

In the distributed system, the existing tasks, ports, and memory addresses are spread over the different nodes. Every node has exactly one kernel, one sender, and one receiver. Figure 5.7 illustrates a distributed system with three nodes, three tasks, two ports, and nine memory addresses.

In the TLA⁺ specification, we will use three parameters, namely *AdrLoc*(-), *TaskLoc*(-), and *PortLoc*(-), to define the location of every memory address, task, and port, respectively. We can assert that each of these elements is given a location writing:

$$\begin{array}{ll} \text{ASSUME } \forall ma \in \text{MemAdr} : \text{AdrLoc}(ma) \in \text{Node} \\ \text{ASSUME } \forall t \in \text{Task} & : \text{TaskLoc}(t) \in \text{Node} \\ \text{ASSUME } \forall p \in \text{Port} & : \text{PortLoc}(p) \in \text{Node} \end{array}$$

The complete specification is module *DistributedCommunication* in Figure 5.8. It grew from the combination of modules *IntertaskCommunication* and *ExternalPacketFlow*. We will go through the specification explaining some new aspects in more detail.

Packets must travel from one node to another. For that, there must be some network connecting the nodes. This is omitted in the specification; it is assumed

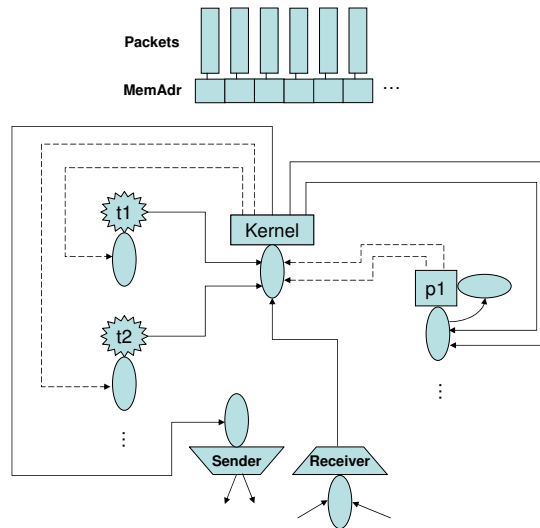


Figure 5.6: Representation of a node.

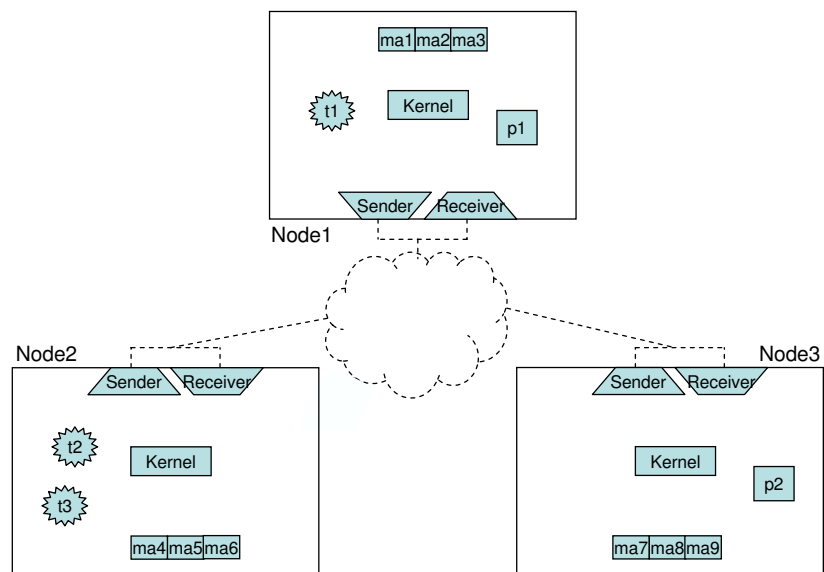


Figure 5.7: Distributed system.

that is possible for a packet to travel from one node to another.⁴ When a packet is exported from one node it is registered in *WireMem*. A packet in *WireMem* can be collected by any node.

Variable *WireMem* is the only “new” variable. All the others had already been used either in module *IntertaskCommunication*, or in module *ExternalPacketFlow*, or in both. Their intuitive meaning remains the same. However, the definition of *KerEP*, *SndEP*, and *RcvEP* is different to before. Because each node possesses a kernel, a sender, and a receiver, these variables are defined as functions that map each node to a subset of memory addresses. For example, variable *KerEP* stays

$$KerEP \in [Node \rightarrow \text{SUBSET} (MemAdr)]$$

The definition of actions *DCIssueRequest*, *DCPortOp*, and *DCTaskOp* is similar to the definition of actions *ICIssueRequest*, *ICPortOp*, and *ICTaskOp*, respectively, of module *IntertaskCommunication*. It is simply necessary to make sure that all the entities are local, namely in the updating of variable *KerEP*.

Actions *DCKernelOp* combines the definition of actions *ICKernelOp* and *EPFKernelOp*. To determine whether a request is local or remote, the kernel must evaluate the location of the destination entity. It is the sole entity that must perform this task. As previously mentioned, all the other entities have a “blind” activity. Yet, the kernel’s definition remains extremely simple.

The definition of actions *DCSndForwardsPacket*, *DCPacketArrives*, and *DCRcvSendsToKernel* derives from the definitions of their “homonym” actions of module *ExternalPacketFlow*. In the first two, it is incorporated the use of variable *WireMem* as explained above.

The safety and liveness requirements are similar to the ones defined in modules *IntertaskCommunication* and *ExternalPacketFlow*.

```

1  MODULE DistributedCommunication
2  CONSTANTS MemAdr, Task, Port, NoPac, Node, AdrLoc(-), TaskLoc(-), PortLoc(-)

4  ASSUME  $\forall ma \in MemAdr : AdrLoc(ma) \in Node$ 
5  ASSUME  $\forall t \in Task : TaskLoc(t) \in Node$ 
6  ASSUME  $\forall p \in Port : PortLoc(p) \in Node$ 

8  VARIABLES Mem, KerEP, PortEP, TReqOk, RcvEP, SndEP, TaskEP, WireMem
9  DCvars  $\triangleq \langle Mem, KerEP, PortEP, TReqOk, RcvEP, SndEP, TaskEP, WireMem \rangle$ 

11 ServType  $\triangleq \{ \text{"sync"}, \text{"async"} \}$ 
12 Service  $\triangleq \{ \text{"orig"}, \text{"comp"}, \text{"acknow"} \}$ 
13 Packet  $\triangleq [src : Task,$ 
14            $dest : Port,$ 
15            $serv : Service,$ 
16            $stype : ServType]$ 

18 DCTypeOk  $\triangleq$ 
19    $\wedge Mem \in [MemAdr \rightarrow Packet \cup \{NoPac\}]$ 
20    $\wedge KerEP \in [Node \rightarrow \text{SUBSET} (MemAdr)]$ 

```

⁴Section 5.3.2 (on page 105) comments this subject a bit further.

```

21   $\wedge PortEP \in [Port \rightarrow [Entry : \text{SUBSET } (MemAdr),$ 
22       $Store : \text{SUBSET } (MemAdr)]]$ 
23   $\wedge TReqOk \in [Task \rightarrow \text{BOOLEAN}]$ 
24   $\wedge RcvEP \in [Node \rightarrow \text{SUBSET } (MemAdr)]$ 
25   $\wedge SndEP \in [Node \rightarrow \text{SUBSET } (MemAdr)]$ 
26   $\wedge TaskEP \in [Task \rightarrow \text{SUBSET } (MemAdr)]$ 
27   $\wedge WireMem \in \text{SUBSET } (Packet)$ 

29   $DCInit \triangleq$ 
30       $\wedge Mem = [ma \in MemAdr \mapsto NoPac]$ 
31       $\wedge KerEP = [n \in Node \mapsto \{\}]$ 
32       $\wedge PortEP = [p \in Port \mapsto [Entry \mapsto \{\},$ 
33           $Store \mapsto \{\}]]$ 
34       $\wedge TReqOk = [t \in Task \mapsto \text{TRUE}]$ 
35       $\wedge RcvEP = [n \in Node \mapsto \{\}]$ 
36       $\wedge SndEP = [n \in Node \mapsto \{\}]$ 
37       $\wedge TaskEP = [t \in Task \mapsto \{\}]$ 
38       $\wedge WireMem = \{\}$ 

39   $DCIssueRequest(ma, t, p, s, st) \triangleq$ 
40       $\wedge TReqOk[t] = \text{TRUE}$ 
41       $\wedge (AdrLoc(ma) = TaskLoc(t) \wedge Mem[ma] = NoPac)$ 
42       $\wedge \text{LET}$ 
43           $n \triangleq TaskLoc(t)$ 
44           $pac \triangleq \text{CHOOSE } paci \in Packet : \wedge paci.src = t$ 
45               $\wedge paci.dest = p$ 
46               $\wedge paci.serv = s$ 
47               $\wedge paci.stype = st$ 
48       $\text{IN}$ 
49           $\wedge Mem' = [Mem \text{ EXCEPT } ![ma] = pac]$ 
50           $\wedge KerEP' = [KerEP \text{ EXCEPT } ![n] = @ \cup \{ma\}]$ 
51           $\wedge TReqOk' = \text{IF } st = \text{"sync"} \text{ THEN } [TReqOk \text{ EXCEPT } ![t] = \text{FALSE}] \text{ ELSE } TReqOk$ 
52           $\wedge \text{UNCHANGED } \langle PortEP, RcvEP, SndEP, TaskEP, WireMem \rangle$ 

55   $DCPortOp(p, ma, xma) \triangleq$ 
56       $\wedge ma \in PortEP[p].Entry$ 
57       $\wedge \text{LET}$ 
58           $n \triangleq PortLoc(p)$ 
59           $pac \triangleq Mem[ma]$ 
60           $s \triangleq pac.serv$ 
61           $DepQ \triangleq PortEP[p].Store$ 
62       $\text{IN}$ 
63           $\text{IF } ((DepQ = \{\}) \vee (\forall mai \in DepQ : Mem[mai].serv = s))$ 
64               $\text{THEN } \text{If } DepQ \text{ is empty or there is no matching request, store the request}$ 
65                   $\wedge PortEP' = [PortEP \text{ EXCEPT } ![p].Store = @ \cup \{ma\},$ 
66                       $![p].Entry = @ \setminus \{ma\}]$ 
67                   $\wedge \text{UNCHANGED } \langle Mem, KerEP \rangle$ 
68               $\text{ELSE } \text{If there is a complementary request, "synchronize"}$ 
69                   $\wedge xma \in PortEP[p].Store$ 
70                   $\wedge PortEP' = [PortEP \text{ EXCEPT } ![p].Store = @ \setminus \{xma\},$ 
71                       $![p].Entry = @ \setminus \{ma\}]$ 
72                   $\wedge Mem' = [Mem \text{ EXCEPT } ![ma].serv = \text{"acknow"},$ 
73                       $![xma].serv = \text{"acknow"}]$ 
74                   $\wedge KerEP' = [KerEP \text{ EXCEPT } ![n] = @ \cup \{ma, xma\}]$ 
75                   $\wedge \text{UNCHANGED } \langle TReqOk, RcvEP, SndEP, TaskEP, WireMem \rangle$ 

77   $DCTaskOp(t, ma) \triangleq$ 
78       $\wedge ma \in TaskEP[t]$ 

```

```

79  ∧ LET
80       $pac \triangleq Mem[ma]$ 
81       $st \triangleq pac.stype$ 
82  IN
83       $TReqOk' = \text{IF } st = \text{"sync"} \text{ THEN } [TReqOk \text{ EXCEPT } ![t] = \text{TRUE}] \text{ ELSE } TReqOk$ 
84       $\wedge Mem' = [Mem \text{ EXCEPT } ![ma] = NoPac]$ 
85       $\wedge TaskEP' = [TaskEP \text{ EXCEPT } ![t] = @ \setminus \{ma\}]$ 
86       $\wedge \text{UNCHANGED } \langle KerEP, PortEP, RcvEP, SndEP, WireMem \rangle$ 
87  |
88   $DCKernelOp(n, ma) \triangleq$ 
89       $\wedge ma \in KerEP[n]$ 
90       $\wedge \text{LET}$ 
91           $pac \triangleq Mem[ma]$ 
92           $p \triangleq pac.dest$ 
93           $t \triangleq pac.src$ 
94           $s \triangleq pac.serv$ 
95           $destnode \triangleq \text{IF } s = \text{"acknow"} \text{ THEN } TaskLoc(t) \text{ ELSE } PortLoc(p)$ 
96      IN
97           $\text{IF } n = destnode$ 
98              THEN local destination
99                   $\wedge \text{IF } s \neq \text{"acknow"}$ 
100                      THEN
101                           $\wedge PortEP' = [PortEP \text{ EXCEPT } ![p].Entry = @ \cup \{ma\}]$ 
102                           $\wedge \text{UNCHANGED } \langle TaskEP \rangle$ 
103                      ELSE s = "acknow"
104                           $\wedge TaskEP' = [TaskEP \text{ EXCEPT } ![t] = @ \cup \{ma\}]$ 
105                           $\wedge \text{UNCHANGED } \langle PortEP \rangle$ 
106                           $\wedge \text{UNCHANGED } \langle SndEP \rangle$ 
107                      ELSE remote destination
108                           $\wedge SndEP' = [SndEP \text{ EXCEPT } ![n] = @ \cup \{ma\}]$ 
109                           $\wedge \text{UNCHANGED } \langle PortEP, TaskEP \rangle$ 
110                   $\wedge KerEP' = [KerEP \text{ EXCEPT } ![n] = @ \setminus \{ma\}]$ 
111                   $\wedge \text{UNCHANGED } \langle Mem, TReqOk, RcvEP, WireMem \rangle$ 
112  |
113   $DCSndForwardsPacket(n, ma) \triangleq$ 
114       $\wedge ma \in SndEP[n]$ 
115       $\wedge WireMem' = WireMem \cup \{Mem[ma]\}$ 
116       $\wedge Mem' = [Mem \text{ EXCEPT } ![ma] = NoPac]$ 
117       $\wedge SndEP' = [SndEP \text{ EXCEPT } ![n] = @ \setminus \{ma\}]$ 
118       $\wedge \text{UNCHANGED } \langle KerEP, PortEP, TReqOk, RcvEP, TaskEP \rangle$ 
119  |
120   $DCPacketArrives(n, ma, pac) \triangleq$ 
121       $\wedge pac \in WireMem$ 
122       $\wedge Mem[ma] = NoPac \wedge AdrLoc(ma) = n$ 
123       $\wedge Mem' = [Mem \text{ EXCEPT } ![ma] = pac]$ 
124       $\wedge RcvEP' = [RcvEP \text{ EXCEPT } ![n] = @ \cup \{ma\}]$ 
125       $\wedge WireMem' = WireMem \setminus \{pac\}$ 
126       $\wedge \text{UNCHANGED } \langle KerEP, PortEP, TReqOk, SndEP, TaskEP \rangle$ 
127  |
128   $DCRcvSendsToKernel(n, ma) \triangleq$ 
129       $\wedge ma \in RcvEP[n]$ 
130       $\wedge KerEP' = [KerEP \text{ EXCEPT } ![n] = @ \cup \{ma\}]$ 
131       $\wedge RcvEP' = [RcvEP \text{ EXCEPT } ![n] = @ \setminus \{ma\}]$ 
132       $\wedge \text{UNCHANGED } \langle Mem, PortEP, TReqOk, SndEP, TaskEP, WireMem \rangle$ 
133  |
134   $DCNext \triangleq$ 
135       $\exists ma \in MemAdr :$ 

```

```

136       $\forall \exists t \in Task :$ 
137       $\exists p \in Port :$ 
138       $\exists s \in Service \setminus \{ "acknow" \} :$ 
139       $\exists st \in ServType :$ 
140       $DCIssueRequest(ma, t, p, s, st)$ 
141       $\forall \exists n \in Node :$ 
142       $\vee DCKernelOp(n, ma)$ 
143       $\vee DCSndForwardsPacket(n, ma)$ 
144       $\vee DCRcvSendsToKernel(n, ma)$ 
145       $\vee \exists pac \in Packet :$ 
146       $DCPacketArrives(n, ma, pac)$ 
147       $\forall \exists t \in Task :$ 
148       $DCTaskOp(t, ma)$ 
149       $\forall \exists p \in Port :$ 
150       $\exists xma \in MemAdr :$ 
151       $DCPortOp(p, ma, xma)$ 

153   $DCSpec \triangleq DCInit \wedge \Box [DCNext]_{DCvars}$ 
154  |-----|
155  LOCAL INSTANCE IntertaskCommunication

157  An address can only be in one EP at a time
158   $DCUniqueSet \triangleq \forall n \in Node : \forall t \in Task : \forall p \in Port :$ 
159   $\forall si \in \{ KerEP[n], RcvEP[n], SndEP[n], TaskEP[t], PortEP[p].Entry, PortEP[p].Store \} :$ 
160   $\forall sj \in \{ KerEP[n], RcvEP[n], SndEP[n], TaskEP[t], PortEP[p].Entry,$ 
161   $PortEP[p].Store \} \setminus \{ si \} :$ 
162   $si \cap sj = \{ \}$ 

164   $DCMemClean \triangleq$  An address is free iff it does not belong do any EP
165   $\forall ma \in MemAdr : Mem[ma] \neq NoPac \equiv$ 
166   $\vee \exists n \in Node :$ 
167   $\vee ma \in KerEP[n]$ 
168   $\vee ma \in RcvEP[n]$ 
169   $\vee ma \in SndEP[n]$ 
170   $\vee \exists p \in Port :$ 
171   $\vee ma \in PortEP[p].Entry$ 
172   $\vee ma \in PortEP[p].Store$ 
173   $\vee \exists t \in Task :$ 
174   $ma \in TaskEP[t]$ 

176   $DCTaskEPOk \triangleq ICTaskEPOk$ 
178   $DCPortEPOk \triangleq ICPortEPOk$ 
180   $DCUniformStore \triangleq ICUniformStore$ 

182   $DCSndEPOk \triangleq$  All packets in SndEP have remote destination
183   $\forall n \in Node :$ 
184   $\forall ma \in SndEP[n] :$ 
185  LET
186   $pac \triangleq Mem[ma]$ 
187   $p \triangleq pac.dest$ 
188   $t \triangleq pac.src$ 
189   $s \triangleq pac.serv$ 
190   $destnode \triangleq$  IF  $s = "acknow"$  THEN  $TaskLoc(t)$  ELSE  $PortLoc(p)$ 
191  IN
192   $n \neq destnode$ 

```

```

194  $DCSafe \triangleq \wedge DCUniqueSet \wedge DCMemClean \wedge DCTaskEPOk$ 
195  $\wedge DCPortEPOk \wedge DCUniformStore \wedge DCSndEPOk$ 
197  $DCMoving \triangleq ICMoving$ 
198

```

Figure 5.8: The *DistributedCommunication* module.

5.3.1 Avoiding deadlocks

The specification in module *DistributedCommunication* is rather “wide”. It is necessary to constrain it to avoid behaviors that might lead to deadlock.

First of all, the system might deadlock because the issued requests do not complement each other. This is analogous to what was described in Section 5.1.3 (page 91) for module *IntertaskCommunication*. Therefore, a similar solution shall be applied.

However, the system might also deadlock due to its distributed nature. In fact, it is possible to create a jam of packets where, even though issued requests actually complement each other, it is not possible for packets to arrive at their final destination. In fact, such scenario can happen when there is no free memory in a node to receive a new packet.⁵ A *sufficient* condition to avoid it is to limit the total number of requests created to the minimum number of memory addresses of a node.

Imposing these restriction in action *MCDIssueRequest* in module *MCDistributedCommunication*, the system no longer deadlocks. The largest instance model-checked was precisely the configuration shown in Figure 5.7. It took TLC approximately two days to model check the system.⁶ The module is shown below.

```

1  ┌────────────────── MODULE MCDistributedCommunication ───────────────────┐
2  EXTENDS DistributedCommunication, Naturals, FiniteSets
4  LOCAL  $RdTS \triangleq \{ti \in Task : TReqOk[ti] = TRUE\}$ 
5  LOCAL  $BusyMem \triangleq \{mai \in MemAdr : Mem[mai] \neq NoPac\}$ 
6  LOCAL  $MAPerNode(n) \triangleq \{mai \in MemAdr : AdrLoc(mai) = n\}$ 
8  ASSUME  $\forall n \in Node : (Cardinality(MAPerNode(n)) > 1)$ 
10  $MCDIssueRequest(ma, t, p, s, st) \triangleq$ 
11  $\wedge DCIssueRequest(ma, t, p, s, st)$ 

```

The number of total requests in the system cannot exceed the number of memory addresses in a node

⁵Note that in a real implementation it should be extremely unlikely to reach such scenario. However, from the moment that there are limited resources, it is a possibility. Please remember that model-checking is not about probabilities; it tell us nothing about the probability of a behavior.

⁶TLC was run on a single processor 1.8 GHz PC running Windows XP.

```

14   $\wedge \forall n \in \text{Node} : (\text{Cardinality}(\text{MAPerNode}(n)) > \text{Cardinality}(\text{BusyMem}))$ 
15   $\wedge \text{WireMem} = \{\}$ 
16   $\wedge \vee$  Either there is still "place" for more than one request
17     $\wedge \forall n \in \text{Node} : (\text{Cardinality}(\text{MAPerNode}(n)) > (\text{Cardinality}(\text{BusyMem}) + 1))$ 
18     $\wedge (\text{Cardinality}(\text{RdTS}) > 1)$ 
19   $\vee$  Or the new request must be complementary to an existent one that still had
20    no complementary request created
21     $\wedge \exists \text{mai} \in \text{MemAdr} : \wedge \text{Mem}[\text{mai}] \neq \text{NoPac}$ 
22       $\wedge \text{Mem}[\text{mai}].\text{dest} = p$ 
23       $\wedge \text{Mem}[\text{mai}].\text{serv} \neq s$ 
24     $\wedge \neg \exists \text{maj} \in \text{MemAdr} : \wedge \text{Mem}[\text{maj}] \neq \text{NoPac}$ 
25       $\wedge \text{Mem}[\text{maj}].\text{dest} = p$ 
26       $\wedge \text{Mem}[\text{maj}].\text{serv} = s$ 
27  

---


28   $\text{MCDNext} \triangleq$ 
29     $\exists \text{ma} \in \text{MemAdr} :$ 
30       $\vee \exists t \in \text{Task} :$ 
31         $\exists p \in \text{Port} :$ 
32           $\exists s \in \text{Service} \setminus \{\text{"acknow"}\} :$ 
33           $\exists \text{st} \in \text{ServType} :$ 
34             $\text{MCDIssueRequest}(\text{ma}, t, p, s, \text{st})$  *
35       $\vee \exists n \in \text{Node} :$ 
36         $\vee \text{DCKernelOp}(n, \text{ma})$ 
37         $\vee \text{DCSndForwardsPacket}(n, \text{ma})$ 
38         $\vee \text{DCRcvSendsToKernel}(n, \text{ma})$ 
39         $\vee \exists \text{pac} \in \text{Packet} :$ 
40           $\text{DCPacketArrives}(n, \text{ma}, \text{pac})$ 
41       $\vee \exists t \in \text{Task} :$ 
42         $\text{DCTaskOp}(t, \text{ma})$ 
43       $\vee \exists p \in \text{Port} :$ 
44         $\exists \text{xma} \in \text{MemAdr} :$ 
45           $\text{DCPortOp}(p, \text{ma}, \text{xma})$ 
46
47   $\text{MCDCLive} \triangleq$ 
48     $\forall \text{ma} \in \text{MemAdr} :$ 
49       $\wedge \forall t \in \text{Task} :$ 
50         $\forall p \in \text{Port} :$ 
51           $\forall s \in \text{Service} \setminus \{\text{"acknow"}\} :$ 
52           $\forall \text{st} \in \text{ServType} :$ 
53             $\text{SF}_{\text{DCvars}}(\text{MCDIssueRequest}(\text{ma}, t, p, s, \text{st}))$ 
54       $\wedge \forall n \in \text{Node} :$ 
55         $\wedge \text{WF}_{\text{DCvars}}(\text{DCKernelOp}(n, \text{ma}))$ 
56         $\wedge \text{WF}_{\text{DCvars}}(\text{DCSndForwardsPacket}(n, \text{ma}))$ 
57         $\wedge \text{WF}_{\text{DCvars}}(\text{DCRcvSendsToKernel}(n, \text{ma}))$ 
58         $\wedge \forall \text{pac} \in \text{Packet} :$ 
59           $\text{SF}_{\text{DCvars}}(\text{DCPacketArrives}(n, \text{ma}, \text{pac}))$ 
60       $\wedge \forall t \in \text{Task} :$ 
61         $\text{WF}_{\text{DCvars}}(\text{DCTaskOp}(t, \text{ma}))$ 
62       $\wedge \forall p \in \text{Port} :$ 
63         $\forall \text{xma} \in \text{MemAdr} :$ 
64           $\text{WF}_{\text{DCvars}}(\text{DCPortOp}(p, \text{ma}, \text{xma}))$ 
65
66   $\text{MCDCSpec} \triangleq \text{DCInit} \wedge \square[\text{MCDNext}]_{\text{DCvars}}$ 
67   $\text{MCDCLiveSpec} \triangleq \text{MCDCSpec} \wedge \text{MCDCLive}$ 
68  

---


69   $\text{MCDNetL} \triangleq \forall n \in \text{Node} : \text{Cardinality}(\text{WireMem}) \leq \text{Cardinality}(\text{MAPerNode}(n))$ 
70  

---



```

71 THEOREM $MCDCSpec \Rightarrow \Box(DCTypeOk \wedge DCSafe \wedge MDCNetL)$
 72 THEOREM $MCDCLiveSpec \Rightarrow DCMoving$
 73

Figure 5.9: The *MCDistributedCommunication* module.

5.3.2 Network Considerations

Module *DistributedCommunication* does not specify details about the connection between the nodes. It is assumed that there is some network that allows packets to travel from one node to another. Let's have a closer look at the assumptions made.

Variable *WireMem* stores every packet “travelling” in the network. Any packet in *WireMem* can be collected by any node. Therefore, a packet does not necessarily go to its final destination node directly. It may even stay for some “time” travelling around other nodes. (The fairness specification asserts that it eventually reaches its final destination.) With the usual meanings, we could have added to the specification of action *DCPacketArrives* the conjunct

$$\begin{array}{l}
 \wedge \text{ LET} \\
 \quad p \triangleq pac.dest \\
 \quad t \triangleq pac.src \\
 \quad s \triangleq pac.serv \\
 \quad destnode \triangleq \text{IF } s = \text{“acknow” THEN } TaskLoc(t) \text{ ELSE } PortLoc(p) \\
 \text{IN} \\
 \quad n = destnode
 \end{array} \tag{5.1}$$

which would force packets to go directly to their final destination node. However, the inclusion of (5.1) would rely on the assumption that every node can be connected to every other node, and disallow behaviors where packets traverse third nodes to reach their final destination. It would be a severe restriction for such a high level specification.⁷

Another aspect to consider is the network maximum load. In fact, in the definition of action *DCSndForwardsPacket*, a sender can always export a packet to the network, i.e., save it in *WireMem*, independently from the number of packets already stored in *WireMem*. It could be advisable to limit this action up to a maximum network load, say *WMax*. So, we could add to the definition of action *DCSndForwardsPacket* the conjunct

$$\wedge Cardinality(WireMem) < WMax \tag{5.2}$$

Because of restrictions imposed in action *MCDCLiveRequest*, the number of packets in *WireMem* is intrinsically limited by the number of memory addresses

⁷Note, however, that the non-inclusion of (5.1) in the specification only prevents us from forbidding packets to traverse third nodes. In rigor, it cannot be concluded that nodes need not to be connected to every other node.

in every node. It can never exceed the minimum number of memory addresses of a node.⁸ As a result, it was chosen not to include (5.2) in the specification.

5.4 Towards a “nano-kernel”

At the beginning of the chapter, it is stated that the models here presented could constitute a basis for the development of “nano-kernel” operating systems. This section briefly addresses that subject.

In reality, a real-time kernel must support several other functionalities not addressed throughout the chapter, as for example timing and clocking aspects. We’ll leave such subject aside.

Communication and synchronization between tasks is assured. On top of that, interrupt handling can also be supported. In fact, an interrupt can simply be processed as an asynchronous request from a “special” task, the interrupt handler. To manage resource access, the requests to lock and release a resource could be implemented as messages sent to a port, corresponding each resource to a specific port. Finally, to support priority based scheduling, every entity input port can be transformed into a prioritized queue.

5.5 Summary

In this chapter, it is devised a mechanism for the communication and synchronization between tasks. The mechanism can be applied to both distributed and non-distributed systems. The design decisions made throughout the chapter allowed the attainment of a solution with a remarkably simple kernel.

Although substantially different in detail, the presented solution shares many guiding principles with the groundings of OpenComRTOS [36].

⁸Invariant *MCDCNetL* asserts this formally.

Chapter 6

Conclusions

This thesis describes the use of TLA^+ /TLC for the development of solutions for real-time operating systems. It were built specifications for resource management, priority inheritance mechanisms, and communication and synchronization primitives. At all times, TLA^+ /TLC showed to be extremely helpful in achieving concise, correct and unambiguous specifications. In fact, some of the modules seem so simple in retrospect that is almost astonishing to recall all the questions and reasoning that preceded them.

TLC model checks TLA^+ specifications. This distinguishes it from most of the other model checkers, which take as input specifications written in low-level languages. TLA^+ is a high-level very expressive language. Hence, combining TLA^+ and TLC allows the validation of high-level descriptions of systems, constituting an excellent starting point for the elaboration of correct programs. It makes developing, prototyping, and debugging systems specifications much more efficient than with a lower-level language. TLA^+ logic is both powerful and simple. Learning TLA^+ should take no longer than learning a widespread programming language such as C#, Java, or C++. TLA^+ is specially suited for the development of concurrent and/or distributed systems. Concurrent programming is difficult. Rigorous reasoning and development are the best way to avoid errors.

Specifications of large systems should be built incrementally. It can be very useful to solve problems and settle key design principles in a gradual manner. Moreover, it is easy to write an erroneous specification and have TLC found no error. Most likely, it lacks some safety and/or liveness properties that would exhibit the error. Large specifications should contain a large set of desired properties. Specifications should include liveness requirements. They are less obvious to write but quite useful to catch errors in a specification.

It is recognized that formal methods can be notably valuable to help producing programs free of design errors. However, in many cases, formal techniques are only applied in the verification and validation of systems after their development. This misdirection of effort leaves space for those software systems to emerge from a set of unclear ideas. Recurring to formal specification, a great

number of potential problems can be detected, analyzed, and solved in the very early stages of the development cycle. Postponing the identification of such problems may highly complicate the task of correcting them. It is so that formal specification has solved much of the computing science problems. Ultimately, it is a choice of moving part of the effort put into testing and maintenance to the initial stages of software development.

It should be noted that, despite its great advantages, formal specification and model-checking is no guarantee that an implementation will be correct. First of all, it is a model that is being checked, not the actual implementation. Then, many specifications can only be checked for instances that are too small to give us complete confidence in their correctness. Model checking should not be advertised as a full verification technique. But, in the end, does such a technique exist? There is a famous quote from Dijkstra: “Testing can never assert the absence of errors; only detect its presence”. And, indeed, model checking is just an advanced testing technique. The writing of hand proofs on the other hand could be considered the ultimate verification technique. But, to be fair, the former Dijkstra quote has a counterpart: “One can never tell that a prove is correct; only that it has found no error in it”. It is not difficult to make a mistake writing a proof. Model checking is not a substitute for everything. Still, it is a very powerful technique. It can increase our confidence in a system. After all, that’s what every technique amounts to: increasing the level of confidence. Absolute certainty is utopian.

This thesis specially addresses concurrency issues. In Chapter 5, it is described a mechanism that is presented as a solution for communication and synchronization between tasks. The mechanism indeed allows the synchronization of tasks. Nevertheless, so does the resource management method developed in Chapter 3. Even though, the term synchronization is never applied throughout this chapter. The same way that (in Chapter 5) we say that tasks “synchronize on a port”, it could also be said (in Chapter 3) that tasks “synchronize on a resource”. Synchronization is not really about resources, ports, or any other term familiar to us (such as monitors and semaphores). It is a fundamental problem. This is to say that one should be able to really understand a system’s behavior. One should be able to abstract away identifiers names and to truly realize what is happening in the system. It is easy to loose such capability getting immersed in an amalgam of different terms and metaphors when, in the end, the set of underlying fundamental principles is actually quite small. Arguably, a great deal of the complexity of software systems rises from here. Abstraction and formal reasoning can greatly contribute to overcome such difficulties. The human brain is still the supreme tool for systems development. TLA⁺ showed to be a fantastic ally.

Appendices

Appendix A

Model-checking Module DistributedInheritance

A.1 Configuration File

As mentioned at the beginning of Section 4.4.1 (page 69), it takes it takes two takes and one shared resource for TLC to find a violation of invariant *DIPrioInh*. Bellow, the configuration file used:

```
(* ..... *)
(*          CONFIG DistributedInheritance.cfg          *)
(* ..... *)

SPECIFICATION  DILiveSpec

CONSTANTS      Resource = {r1}
                Task = {t1, t2}
                NoTask = NoTask
                NoRes = NoRes
                PriMax = 3
                QMax = 1

INVARIANT      DTypeOk SRMNotification IRALock DISecCorrect DIPrioInh

PROPERTY       SRMSpec

(* ..... *)
```

A.2 Error Trace

The violation of invariant *DIPrioInh* is found after 22 steps. Bellow, the error trace reported by TLC:

```
TLC Version 2.0 of January 16, 2006
Model-checking
Parsing file DistributedInheritance.tla
[...]
Semantic processing of module DistributedInheritance

Finished computing initial states: 9 states generated, with 6 of them
distinct.
Error: Invariant DIPrioInh is violated.
The behavior up to this point is:

STATE 1: <Initial predicate>
^ TNominalPri = (t1 :> 1 @@ t2 :> 2)
^ SecActPriQ = << >>
^ ResState = (r1 :> [Locked |-> FALSE, WL |-> , Owner |-> NoTask])
^ ConsQ = << >>
^ SecResState = (r1 :> [Locked |-> FALSE, WL |-> , Owner |-> NoTask])
^ ResNotify = (r1 :> FALSE)
^ GetQ = << >>
^ PGlobQ = << >>
^ SecTaskOw = ( t1 :> [Owning |-> , WaitsR |-> NoRes] @@
t2 :> [Owning |-> , WaitsR |-> NoRes] )
^ TaskState = (t1 :> "Ready" @@ t2 :> "Ready")
^ TActivePri = (t1 :> 1 @@ t2 :> 2)
^ SecTActivePri = (t1 :> 1 @@ t2 :> 2)
^ TaskOw = ( t1 :> [Owning |-> , WaitsR |-> NoRes] @@
t2 :> [Owning |-> , WaitsR |-> NoRes] )

STATE 2: <Action line 49, col 5 to line 51, col 60 of module
DistributedInheritance>
^ TNominalPri = (t1 :> 1 @@ t2 :> 2)
^ SecActPriQ = << >>
^ ResState = (r1 :> [Locked |-> FALSE, WL |-> , Owner |-> NoTask])
^ ConsQ = << >>
^ SecResState = (r1 :> [Locked |-> FALSE, WL |-> , Owner |-> NoTask])
^ ResNotify = (r1 :> FALSE)
^ GetQ = << >>
^ PGlobQ = << >>
^ SecTaskOw = ( t1 :> [Owning |-> , WaitsR |-> NoRes] @@
t2 :> [Owning |-> , WaitsR |-> NoRes] )
^ TaskState = (t1 :> "Running" @@ t2 :> "Ready")
^ TActivePri = (t1 :> 1 @@ t2 :> 2)
^ SecTActivePri = (t1 :> 1 @@ t2 :> 2)
^ TaskOw = ( t1 :> [Owning |-> , WaitsR |-> NoRes] @@
t2 :> [Owning |-> , WaitsR |-> NoRes] )
```

```

STATE 3: <Action line 54, col 5 to line 86, col 86 of module
DistributedInheritance>
^ TNominalPri = (t1 :> 1 @@ t2 :> 2)
^ SecActPriQ = << >>
^ ResState = (r1 :> [Locked |-> TRUE, WL |-> , Owner |-> t1])
^ ConsQ = << >>
^ SecResState = (r1 :> [Locked |-> FALSE, WL |-> , Owner |-> NoTask])
^ ResNotify = (r1 :> FALSE)
^ GetQ = << >>
^ PGlobQ = << [Type |-> "TLocksR", OnTask |-> t1, OnRes |-> r1] >>
^ SecTaskOw = ( t1 :> [Owning |-> , WaitsR |-> NoRes] @@
t2 :> [Owning |-> , WaitsR |-> NoRes] )
^ TaskState = (t1 :> "Running" @@ t2 :> "Ready")
^ TActivePri = (t1 :> 1 @@ t2 :> 2)
^ SecTActivePri = (t1 :> 1 @@ t2 :> 2)
^ TaskOw = ( t1 :> [Owning |-> r1, WaitsR |-> NoRes] @@
t2 :> [Owning |-> , WaitsR |-> NoRes] )

STATE 4: <Action line 49, col 5 to line 51, col 60 of module
DistributedInheritance>
^ TNominalPri = (t1 :> 1 @@ t2 :> 2)
^ SecActPriQ = << >>
^ ResState = (r1 :> [Locked |-> TRUE, WL |-> , Owner |-> t1])
^ ConsQ = << >>
^ SecResState = (r1 :> [Locked |-> FALSE, WL |-> , Owner |-> NoTask])
^ ResNotify = (r1 :> FALSE)
^ GetQ = << >>
^ PGlobQ = << [Type |-> "TLocksR", OnTask |-> t1, OnRes |-> r1] >>
^ SecTaskOw = ( t1 :> [Owning |-> , WaitsR |-> NoRes] @@
t2 :> [Owning |-> , WaitsR |-> NoRes] )
^ TaskState = (t1 :> "Running" @@ t2 :> "Running")
^ TActivePri = (t1 :> 1 @@ t2 :> 2)
^ SecTActivePri = (t1 :> 1 @@ t2 :> 2)
^ TaskOw = ( t1 :> [Owning |-> r1, WaitsR |-> NoRes] @@
t2 :> [Owning |-> , WaitsR |-> NoRes] )

STATE 5: <Action line 133, col 5 to line 164, col 77 of module
DistributedInheritance>
^ TNominalPri = (t1 :> 1 @@ t2 :> 2)
^ SecActPriQ = << >>
^ ResState = (r1 :> [Locked |-> TRUE, WL |-> , Owner |-> t1])
^ ConsQ = << [Type |-> "TLocksR", OnTask |-> t1, OnRes |-> r1] >>
^ SecResState = (r1 :> [Locked |-> TRUE, WL |-> , Owner |-> t1])
^ ResNotify = (r1 :> FALSE)
^ GetQ = << >>
^ PGlobQ = << >>
^ SecTaskOw = ( t1 :> [Owning |-> r1, WaitsR |-> NoRes] @@
t2 :> [Owning |-> , WaitsR |-> NoRes] )
^ TaskState = (t1 :> "Running" @@ t2 :> "Running")
^ TActivePri = (t1 :> 1 @@ t2 :> 2)
^ SecTActivePri = (t1 :> 1 @@ t2 :> 2)
^ TaskOw = ( t1 :> [Owning |-> r1, WaitsR |-> NoRes] @@
t2 :> [Owning |-> , WaitsR |-> NoRes] )

```

```

STATE 6: <Action line 54, col 5 to line 86, col 86 of module
DistributedInheritance>
^ TNominalPri = (t1 :> 1 @@ t2 :> 2)
^ SecActPriQ = << >>
^ ResState = (r1 :> [Locked |-> TRUE, WL |-> t2, Owner |-> t1])
^ ConsQ = << [Type |-> "TLocksR", OnTask |-> t1, OnRes |-> r1] >>
^ SecResState = (r1 :> [Locked |-> TRUE, WL |-> , Owner |-> t1])
^ ResNotify = (r1 :> FALSE)
^ GetQ = << >>
^ PGlobQ = << [Type |-> "TWaitsOnR", OnTask |-> t2, OnRes |-> r1] >>
^ SecTaskOw = ( t1 :> [Owning |-> r1, WaitsR |-> NoRes] @@
t2 :> [Owning |-> , WaitsR |-> NoRes] )
^ TaskState = (t1 :> "Running" @@ t2 :> "Waiting")
^ TActivePri = (t1 :> 1 @@ t2 :> 2)
^ SecTActivePri = (t1 :> 1 @@ t2 :> 2)
^ TaskOw = ( t1 :> [Owning |-> r1, WaitsR |-> NoRes] @@
t2 :> [Owning |-> , WaitsR |-> r1] )

```

```

STATE 7: <Action line 167, col 5 to line 222, col 94 of module
DistributedInheritance>
^ TNominalPri = (t1 :> 1 @@ t2 :> 2)
^ SecActPriQ = << >>
^ ResState = (r1 :> [Locked |-> TRUE, WL |-> t2, Owner |-> t1])
^ ConsQ = << >>
^ SecResState = (r1 :> [Locked |-> TRUE, WL |-> , Owner |-> t1])
^ ResNotify = (r1 :> FALSE)
^ GetQ = << >>
^ PGlobQ = << [Type |-> "TWaitsOnR", OnTask |-> t2, OnRes |-> r1] >>
^ SecTaskOw = ( t1 :> [Owning |-> r1, WaitsR |-> NoRes] @@
t2 :> [Owning |-> , WaitsR |-> NoRes] )
^ TaskState = (t1 :> "Running" @@ t2 :> "Waiting")
^ TActivePri = (t1 :> 1 @@ t2 :> 2)
^ SecTActivePri = (t1 :> 1 @@ t2 :> 2)
^ TaskOw = ( t1 :> [Owning |-> r1, WaitsR |-> NoRes] @@
t2 :> [Owning |-> , WaitsR |-> r1] )

```

```

STATE 8: <Action line 133, col 5 to line 164, col 77 of module
DistributedInheritance>
^ TNominalPri = (t1 :> 1 @@ t2 :> 2)
^ SecActPriQ = << >>
^ ResState = (r1 :> [Locked |-> TRUE, WL |-> t2, Owner |-> t1])
^ ConsQ = << [Type |-> "TWaitsOnR", OnTask |-> t2, OnRes |-> r1] >>
^ SecResState = (r1 :> [Locked |-> TRUE, WL |-> t2, Owner |-> t1])
^ ResNotify = (r1 :> FALSE)
^ GetQ = << >>
^ PGlobQ = << >>
^ SecTaskOw = ( t1 :> [Owning |-> r1, WaitsR |-> NoRes] @@
t2 :> [Owning |-> , WaitsR |-> r1] )
^ TaskState = (t1 :> "Running" @@ t2 :> "Waiting")
^ TActivePri = (t1 :> 1 @@ t2 :> 2)
^ SecTActivePri = (t1 :> 1 @@ t2 :> 2)
^ TaskOw = ( t1 :> [Owning |-> r1, WaitsR |-> NoRes] @@
t2 :> [Owning |-> , WaitsR |-> r1] )

```

```

STATE 9: <Action line 89, col 5 to line 99, col 78 of module
DistributedInheritance>
^ TNominalPri = (t1 :> 1 @@ t2 :> 2)
^ SecActPriQ = << >>
^ ResState = (r1 :> [Locked |-> FALSE, WL |-> t2, Owner |-> NoTask])
^ ConsQ = << [Type |-> "TWaitsOnR", OnTask |-> t2, OnRes |-> r1] >>
^ SecResState = (r1 :> [Locked |-> TRUE, WL |-> t2, Owner |-> t1])
^ ResNotify = (r1 :> TRUE)
^ GetQ = << >>
^ PGlobQ = << [Type |-> "TReleasesR", OnTask |-> t1, OnRes |-> r1] >>
^ SecTaskOw = ( t1 :> [Owning |-> r1, WaitsR |-> NoRes] @@
t2 :> [Owning |-> , WaitsR |-> r1] )
^ TaskState = (t1 :> "Running" @@ t2 :> "Waiting")
^ TActivePri = (t1 :> 1 @@ t2 :> 2)
^ SecTActivePri = (t1 :> 1 @@ t2 :> 2)
^ TaskOw = ( t1 :> [Owning |-> , WaitsR |-> NoRes] @@
t2 :> [Owning |-> , WaitsR |-> r1] )

STATE 10: <Action line 167, col 5 to line 222, col 94 of module
DistributedInheritance>
^ TNominalPri = (t1 :> 1 @@ t2 :> 2)
^ SecActPriQ = << (t1 :> 2 @@ t2 :> 2) >>
^ ResState = (r1 :> [Locked |-> FALSE, WL |-> t2, Owner |-> NoTask])
^ ConsQ = << >>
^ SecResState = (r1 :> [Locked |-> TRUE, WL |-> t2, Owner |-> t1])
^ ResNotify = (r1 :> TRUE)
^ GetQ = << >>
^ PGlobQ = << [Type |-> "TReleasesR", OnTask |-> t1, OnRes |-> r1] >>
^ SecTaskOw = ( t1 :> [Owning |-> r1, WaitsR |-> NoRes] @@
t2 :> [Owning |-> , WaitsR |-> r1] )
^ TaskState = (t1 :> "Running" @@ t2 :> "Waiting")
^ TActivePri = (t1 :> 1 @@ t2 :> 2)
^ SecTActivePri = (t1 :> 2 @@ t2 :> 2)
^ TaskOw = ( t1 :> [Owning |-> , WaitsR |-> NoRes] @@
t2 :> [Owning |-> , WaitsR |-> r1] )

STATE 11: <Action line 133, col 5 to line 164, col 77 of module
DistributedInheritance>
^ TNominalPri = (t1 :> 1 @@ t2 :> 2)
^ SecActPriQ = << (t1 :> 2 @@ t2 :> 2) >>
^ ResState = (r1 :> [Locked |-> FALSE, WL |-> t2, Owner |-> NoTask])
^ ConsQ = << [Type |-> "TReleasesR", OnTask |-> t1, OnRes |-> r1] >>
^ SecResState = (r1 :> [Locked |-> FALSE, WL |-> t2, Owner |-> NoTask])
^ ResNotify = (r1 :> TRUE)
^ GetQ = << >>
^ PGlobQ = << >>
^ SecTaskOw = ( t1 :> [Owning |-> , WaitsR |-> NoRes] @@
t2 :> [Owning |-> , WaitsR |-> r1] )
^ TaskState = (t1 :> "Running" @@ t2 :> "Waiting")
^ TActivePri = (t1 :> 1 @@ t2 :> 2)
^ SecTActivePri = (t1 :> 2 @@ t2 :> 2)
^ TaskOw = ( t1 :> [Owning |-> , WaitsR |-> NoRes] @@
t2 :> [Owning |-> , WaitsR |-> r1] )

```

```

STATE 12: <Action line 54, col 5 to line 86, col 86 of module
DistributedInheritance>
^ TNominalPri = (t1 :> 1 @@ t2 :> 2)
^ SecActPriQ = << (t1 :> 2 @@ t2 :> 2) >>
^ ResState = (r1 :> [Locked |-> FALSE, WL |-> t1, t2, Owner |-> NoTask])
^ ConsQ = << [Type |-> "TReleasesR", OnTask |-> t1, OnRes |-> r1] >>
^ SecResState = (r1 :> [Locked |-> FALSE, WL |-> t2, Owner |-> NoTask])
^ ResNotify = (r1 :> TRUE)
^ GetQ = << >>
^ PGlobQ = << [Type |-> "TWaitsOnR", OnTask |-> t1, OnRes |-> r1] >>
^ SecTaskOw = ( t1 :> [Owning |-> , WaitsR |-> NoRes] @@
t2 :> [Owning |-> , WaitsR |-> r1] )
^ TaskState = (t1 :> "Waiting" @@ t2 :> "Waiting")
^ TActivePri = (t1 :> 1 @@ t2 :> 2)
^ SecTActivePri = (t1 :> 2 @@ t2 :> 2)
^ TaskOw = (t1 :> [Owning |-> , WaitsR |-> r1] @@ t2 :> [Owning |-> ,
WaitsR |-> r1])

```

```

STATE 13: <Action line 225, col 5 to line 230, col 81 of module
DistributedInheritance>
^ TNominalPri = (t1 :> 1 @@ t2 :> 2)
^ SecActPriQ = << >>
^ ResState = (r1 :> [Locked |-> FALSE, WL |-> t1, t2, Owner |-> NoTask])
^ ConsQ = << [Type |-> "TReleasesR", OnTask |-> t1, OnRes |-> r1] >>
^ SecResState = (r1 :> [Locked |-> FALSE, WL |-> t2, Owner |-> NoTask])
^ ResNotify = (r1 :> TRUE)
^ GetQ = << (t1 :> 2 @@ t2 :> 2) >>
^ PGlobQ = << [Type |-> "TWaitsOnR", OnTask |-> t1, OnRes |-> r1] >>
^ SecTaskOw = ( t1 :> [Owning |-> , WaitsR |-> NoRes] @@
t2 :> [Owning |-> , WaitsR |-> r1] )
^ TaskState = (t1 :> "Waiting" @@ t2 :> "Waiting")
^ TActivePri = (t1 :> 1 @@ t2 :> 2)
^ SecTActivePri = (t1 :> 2 @@ t2 :> 2)
^ TaskOw = (t1 :> [Owning |-> , WaitsR |-> r1] @@ t2 :> [Owning |-> ,
WaitsR |-> r1])

```

```

STATE 14: <Action line 167, col 5 to line 222, col 94 of module
DistributedInheritance>
^ TNominalPri = (t1 :> 1 @@ t2 :> 2)
^ SecActPriQ = << (t1 :> 1 @@ t2 :> 2) >>
^ ResState = (r1 :> [Locked |-> FALSE, WL |-> t1, t2, Owner |-> NoTask])
^ ConsQ = << >>
^ SecResState = (r1 :> [Locked |-> FALSE, WL |-> t2, Owner |-> NoTask])
^ ResNotify = (r1 :> TRUE)
^ GetQ = << (t1 :> 2 @@ t2 :> 2) >>
^ PGlobQ = << [Type |-> "TWaitsOnR", OnTask |-> t1, OnRes |-> r1] >>
^ SecTaskOw = ( t1 :> [Owning |-> , WaitsR |-> NoRes] @@
t2 :> [Owning |-> , WaitsR |-> r1] )
^ TaskState = (t1 :> "Waiting" @@ t2 :> "Waiting")
^ TActivePri = (t1 :> 1 @@ t2 :> 2)
^ SecTActivePri = (t1 :> 1 @@ t2 :> 2)
^ TaskOw = (t1 :> [Owning |-> , WaitsR |-> r1] @@ t2 :> [Owning |-> ,
WaitsR |-> r1])

```

```

STATE 15: <Action line 133, col 5 to line 164, col 77 of module
DistributedInheritance>
^ TNominalPri = (t1 :> 1 @@ t2 :> 2)
^ SecActPriQ = << (t1 :> 1 @@ t2 :> 2) >>
^ ResState = (r1 :> [Locked |-> FALSE, WL |-> t1, t2, Owner |-> NoTask])
^ ConsQ = << [Type |-> "TWaitsOnR", OnTask |-> t1, OnRes |-> r1] >>
^ SecResState = (r1 :> [Locked |-> FALSE, WL |-> t1, t2, Owner |-> NoTask])
^ ResNotify = (r1 :> TRUE)
^ GetQ = << (t1 :> 2 @@ t2 :> 2) >>
^ PGlobQ = << >>
^ SecTaskOw = (t1 :> [Owning |-> , WaitsR |-> r1] @@ t2 :> [Owning |-> ,
WaitsR |-> r1])
^ TaskState = (t1 :> "Waiting" @@ t2 :> "Waiting")
^ TActivePri = (t1 :> 1 @@ t2 :> 2)
^ SecTActivePri = (t1 :> 1 @@ t2 :> 2)
^ TaskOw = (t1 :> [Owning |-> , WaitsR |-> r1] @@ t2 :> [Owning |-> ,
WaitsR |-> r1])

STATE 16: <Action line 167, col 5 to line 222, col 94 of module
DistributedInheritance>
^ TNominalPri = (t1 :> 1 @@ t2 :> 2)
^ SecActPriQ = << (t1 :> 1 @@ t2 :> 2) >>
^ ResState = (r1 :> [Locked |-> FALSE, WL |-> t1, t2, Owner |-> NoTask])
^ ConsQ = << >>
^ SecResState = (r1 :> [Locked |-> FALSE, WL |-> t1, t2, Owner |-> NoTask])
^ ResNotify = (r1 :> TRUE)
^ GetQ = << (t1 :> 2 @@ t2 :> 2) >>
^ PGlobQ = << >>
^ SecTaskOw = (t1 :> [Owning |-> , WaitsR |-> r1] @@ t2 :> [Owning |-> ,
WaitsR |-> r1])
^ TaskState = (t1 :> "Waiting" @@ t2 :> "Waiting")
^ TActivePri = (t1 :> 1 @@ t2 :> 2)
^ SecTActivePri = (t1 :> 1 @@ t2 :> 2)
^ TaskOw = (t1 :> [Owning |-> , WaitsR |-> r1] @@ t2 :> [Owning |-> ,
WaitsR |-> r1])

STATE 17: <Action line 233, col 5 to line 237, col 80 of module
DistributedInheritance>
^ TNominalPri = (t1 :> 1 @@ t2 :> 2)
^ SecActPriQ = << (t1 :> 1 @@ t2 :> 2) >>
^ ResState = (r1 :> [Locked |-> FALSE, WL |-> t1, t2, Owner |-> NoTask])
^ ConsQ = << >>
^ SecResState = (r1 :> [Locked |-> FALSE, WL |-> t1, t2, Owner |-> NoTask])
^ ResNotify = (r1 :> TRUE)
^ GetQ = << >>
^ PGlobQ = << >>
^ SecTaskOw = (t1 :> [Owning |-> , WaitsR |-> r1] @@ t2 :> [Owning |-> ,
WaitsR |-> r1])
^ TaskState = (t1 :> "Waiting" @@ t2 :> "Waiting")
^ TActivePri = (t1 :> 2 @@ t2 :> 2)
^ SecTActivePri = (t1 :> 1 @@ t2 :> 2)
^ TaskOw = (t1 :> [Owning |-> , WaitsR |-> r1] @@ t2 :> [Owning |-> ,
WaitsR |-> r1])

```

```

STATE 18: <Action line 102, col 5 to line 125, col 79 of module
DistributedInheritance>
^ TNominalPri = (t1 :> 1 @@ t2 :> 2)
^ SecActPriQ = << (t1 :> 1 @@ t2 :> 2) >>
^ ResState = (r1 :> [Locked |-> TRUE, WL |-> t2, Owner |-> t1])
^ ConsQ = << >>
^ SecResState = (r1 :> [Locked |-> FALSE, WL |-> t1, t2, Owner |-> NoTask])
^ ResNotify = (r1 :> FALSE)
^ GetQ = << >>
^ PGlobQ = << [Type |-> "TIsNotified", OnTask |-> t1, OnRes |-> r1] >>
^ SecTaskOw = (t1 :> [Owning |-> , WaitsR |-> r1] @@ t2 :> [Owning |-> ,
WaitsR |-> r1])
^ TaskState = (t1 :> "Ready" @@ t2 :> "Waiting")
^ TActivePri = (t1 :> 2 @@ t2 :> 2)
^ SecTActivePri = (t1 :> 1 @@ t2 :> 2)
^ TaskOw = ( t1 :> [Owning |-> r1, WaitsR |-> NoRes] @@
t2 :> [Owning |-> , WaitsR |-> r1] )

```

```

STATE 19: <Action line 133, col 5 to line 164, col 77 of module
DistributedInheritance>
^ TNominalPri = (t1 :> 1 @@ t2 :> 2)
^ SecActPriQ = << (t1 :> 1 @@ t2 :> 2) >>
^ ResState = (r1 :> [Locked |-> TRUE, WL |-> t2, Owner |-> t1])
^ ConsQ = << [Type |-> "TIsNotified", OnTask |-> t1, OnRes |-> r1] >>
^ SecResState = (r1 :> [Locked |-> TRUE, WL |-> t2, Owner |-> t1])
^ ResNotify = (r1 :> FALSE)
^ GetQ = << >>
^ PGlobQ = << >>
^ SecTaskOw = ( t1 :> [Owning |-> r1, WaitsR |-> NoRes] @@
t2 :> [Owning |-> , WaitsR |-> r1] )
^ TaskState = (t1 :> "Ready" @@ t2 :> "Waiting")
^ TActivePri = (t1 :> 2 @@ t2 :> 2)
^ SecTActivePri = (t1 :> 1 @@ t2 :> 2)
^ TaskOw = ( t1 :> [Owning |-> r1, WaitsR |-> NoRes] @@
t2 :> [Owning |-> , WaitsR |-> r1] )

```

```

STATE 20: <Action line 167, col 5 to line 222, col 94 of module
DistributedInheritance>
^ TNominalPri = (t1 :> 1 @@ t2 :> 2)
^ SecActPriQ = << (t1 :> 1 @@ t2 :> 2) >>
^ ResState = (r1 :> [Locked |-> TRUE, WL |-> t2, Owner |-> t1])
^ ConsQ = << >>
^ SecResState = (r1 :> [Locked |-> TRUE, WL |-> t2, Owner |-> t1])
^ ResNotify = (r1 :> FALSE)
^ GetQ = << >>
^ PGlobQ = << >>
^ SecTaskOw = ( t1 :> [Owning |-> r1, WaitsR |-> NoRes] @@
t2 :> [Owning |-> , WaitsR |-> r1] )
^ TaskState = (t1 :> "Ready" @@ t2 :> "Waiting")
^ TActivePri = (t1 :> 2 @@ t2 :> 2)
^ SecTActivePri = (t1 :> 1 @@ t2 :> 2)
^ TaskOw = ( t1 :> [Owning |-> r1, WaitsR |-> NoRes] @@
t2 :> [Owning |-> , WaitsR |-> r1] )

```

```

STATE 21: <Action line 225, col 5 to line 230, col 81 of module
DistributedInheritance>
^ TNominalPri = (t1 :> 1 @@ t2 :> 2)
^ SecActPriQ = << >>
^ ResState = (r1 :> [Locked |-> TRUE, WL |-> t2, Owner |-> t1])
^ ConsQ = << >>
^ SecResState = (r1 :> [Locked |-> TRUE, WL |-> t2, Owner |-> t1])
^ ResNotify = (r1 :> FALSE)
^ GetQ = << (t1 :> 1 @@ t2 :> 2) >>
^ PGlobQ = << >>
^ SecTaskOw = ( t1 :> [Owning |-> r1, WaitsR |-> NoRes] @@
t2 :> [Owning |-> , WaitsR |-> r1] )
^ TaskState = (t1 :> "Ready" @@ t2 :> "Waiting")
^ TActivePri = (t1 :> 2 @@ t2 :> 2)
^ SecTActivePri = (t1 :> 1 @@ t2 :> 2)
^ TaskOw = ( t1 :> [Owning |-> r1, WaitsR |-> NoRes] @@
t2 :> [Owning |-> , WaitsR |-> r1] )

STATE 22: <Action line 233, col 5 to line 237, col 80 of module
DistributedInheritance>
^ TNominalPri = (t1 :> 1 @@ t2 :> 2)
^ SecActPriQ = << >>
^ ResState = (r1 :> [Locked |-> TRUE, WL |-> t2, Owner |-> t1])
^ ConsQ = << >>
^ SecResState = (r1 :> [Locked |-> TRUE, WL |-> t2, Owner |-> t1])
^ ResNotify = (r1 :> FALSE)
^ GetQ = << >>
^ PGlobQ = << >>
^ SecTaskOw = ( t1 :> [Owning |-> r1, WaitsR |-> NoRes] @@
t2 :> [Owning |-> , WaitsR |-> r1] )
^ TaskState = (t1 :> "Ready" @@ t2 :> "Waiting")
^ TActivePri = (t1 :> 1 @@ t2 :> 2)
^ SecTActivePri = (t1 :> 1 @@ t2 :> 2)
^ TaskOw = ( t1 :> [Owning |-> r1, WaitsR |-> NoRes] @@
t2 :> [Owning |-> , WaitsR |-> r1] )

13972 states generated, 3932 distinct states found, 268 states left
on queue.
The depth of the complete state graph search is 22.

```


Bibliography

- [1] Martín Abadi and Leslie Lamport. The existence of refinement mappings. *Theor. Comput. Sci.*, 82(2):253–284, 1991.
- [2] Michael J. Accetta, Robert V. Baron, William J. Bolosky, David B. Golub, Richard F. Rashid, Avadis Tevanian, and Michael Young. Mach: A New Kernel Foundation for UNIX Development. In *USENIX Summer*, pages 93–113, 1986.
- [3] Alfred V. Aho, M. R. Garey, and Jeffrey D. Ullman. The transitive reduction of a directed graph. *SIAM J. Comput.*, 1(2):131–137, 1972.
- [4] T. P. Baker. Stack-based scheduling for realtime processes. *Real-Time Syst.*, 3(1):67–99, 1991.
- [5] Raymond T. Boute. Calculational semantics: Deriving programming theories from equations by functional predicate calculus. *ACM Trans. Program. Lang. Syst.*, 28(4):747–793, 2006.
- [6] Giorgio C. Buttazzo. *Hard Real-Time Computing Systems: Predictable Scheduling Algorithms and Applications*. Kluwer, 1997.
- [7] Giorgio C. Buttazzo. Rate Monotonic vs. EDF: Judgment Day. *Real-Time Syst.*, 29(1):5–26, 2005.
- [8] Min-Ih Chen and Kwei-Jay Lin. Dynamic priority ceilings: a concurrency control protocol for real-time systems. *Real-Time Syst.*, 2(4):325–346, 1990.
- [9] Edmund M. Clarke and E. Allen Emerson. Design and Synthesis of Synchronization Skeletons Using Branching-Time Temporal Logic. In *Logic of Programs, Workshop*, pages 52–71, London, UK, 1982. Springer-Verlag.
- [10] C. A. R. Hoare. *Communicating sequential processes*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1985.
- [11] Gerard Holzmann. *The SPIN Model Checker: Primer and Reference Manual*. Pearson Education, 2003.

- [12] Kevin Jeffay. Scheduling sporadic tasks with shared resources in hard-real-time systems. In *IEEE Real-Time Systems Symposium*, pages 89–99, 1992.
- [13] Leslie Lamport. Solved problems, unsolved problems and non-problems in concurrency. In *PODC '84: Proceedings of the third annual ACM symposium on Principles of distributed computing*, pages 1–11, New York, NY, USA, 1984. ACM Press.
- [14] Leslie Lamport. The temporal logic of actions. *ACM Trans. Program. Lang. Syst.*, 16(3):872–923, 1994.
- [15] Leslie Lamport. *Specifying Systems: The TLA+ Language and Tools for Hardware and Software Engineers*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2002.
- [16] Leslie Lamport. Real-time model checking is really simple. In Dominique Borriane and Wolfgang J. Paul, editors, *CHARME*, volume 3725 of *Lecture Notes in Computer Science*, pages 162–175. Springer, 2005.
- [17] Leslie Lamport. Checking a multithreaded algorithm with $^+$ cal. In Shlomi Dolev, editor, *DISC*, volume 4167 of *Lecture Notes in Computer Science*, pages 151–163. Springer, 2006.
- [18] Leslie Lamport and Lawrence C. Paulson. Should your specification language be typed. *ACM Trans. Program. Lang. Syst.*, 21(3):502–526, 1999.
- [19] Jean-Claude Laprie and Brian Randell. Basic concepts and taxonomy of dependable and secure computing. *IEEE Trans. Dependable Secur. Comput.*, 1(1):11–33, 2004. Fellow-Algirdas Avizienis and Senior Member-Carl Landwehr.
- [20] Nancy Leveson. *Safeware : System Safety and Computers*. Addison-Wesley Professional, April 1995.
- [21] C. L. Liu and James W. Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. *J. ACM*, 20(1):46–61, 1973.
- [22] J. Ian Munro. Efficient determination of the transitive closure of a directed graph. *Inf. Process. Lett.*, 1(2):56–58, 1971.
- [23] Jean-Pierre Queille and Joseph Sifakis. Specification and verification of concurrent systems in CESAR. In *Proceedings of the 5th Colloquium on International Symposium on Programming*, pages 337–351, London, UK, 1982. Springer-Verlag.
- [24] Glenn E. Reeves. What really happened on Mars. <http://flint.cs.yale.edu/cs422/doc/mars.ps>, 1997.
- [25] A. W. Roscoe, C. A. R. Hoare, and Richard Bird. *The Theory and Practice of Concurrency*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 1997.

- [26] Theo C. Ruys. Towards Effective Model Checking. PhD-Thesis, 2001.
- [27] L. Sha, R. Rajkumar, and J. P. Lehoczky. Priority Inheritance Protocols: An Approach to Real-Time Synchronization. *IEEE Trans. Comput.*, 39(9):1175–1185, 1990.
- [28] L. Sha, R. Rajkumar, and J. P. Lehoczky. Priority inheritance protocols: An approach to real-time synchronization. *IEEE Trans. Comput.*, 39(9):1175–1185, 1990.
- [29] Brinkley Sprunt, Lui Sha, and John P. Lehoczky. Aperiodic task scheduling for hard real-time systems. *Real-Time Systems*, 1(1):27–60, 1989.
- [30] M. Spuri, G. Buttazzo, and F. Sensini. Robust aperiodic scheduling under dynamic priority systems. In *RTSS '95: Proceedings of the 16th IEEE Real-Time Systems Symposium (RTSS '95)*, page 210, Washington, DC, USA, 1995. IEEE Computer Society.
- [31] J. Stankovic and K. Ramamritham. Tutorial on hard real-time systems. IEEE Computer Society Press, 1988.
- [32] John A. Stankovic, Krithi Ramamritham, and Marco Spuri. *Deadline Scheduling for Real-Time Systems: Edf and Related Algorithms*. Kluwer Academic Publishers, Norwell, MA, USA, 1998.
- [33] Jay K. Strosnider, John P. Lehoczky, and Lui Sha. The Deferrable Server Algorithm for Enhanced Aperiodic Responsiveness in Hard Real-Time Environments. *IEEE Trans. Comput.*, 44(1):73–91, 1995.
- [34] Michael M. Swift, Muthukaruppan Annamalai, Brian N. Bershad, and Henry M. Levy. Recovering device drivers. In *OSDI'04: Proceedings of the 6th conference on Symposium on Operating Systems Design & Implementation*, pages 1–1, Berkeley, CA, USA, 2004. USENIX Association.
- [35] Azul Systems. Web page. <http://www.azulsystems.com>.
- [36] Eric Verhulst and Gjalt G. de Jong. OpenComRTOS: An Ultra-Small Network Centric Embedded RTOS Designed Using Formal Modeling. In Emmanuel Gaudin, Elie Najm, and Rick Reed, editors, *SDL Forum*, volume 4745 of *Lecture Notes in Computer Science*, pages 258–271. Springer, 2007.
- [37] Qiwen W. Xu. On compositionality in refining concurrent systems. In J.-F. He, editor, *Proceedings in BCS FACS 7th Refinement Workshop*, Bath, U.K., 1996. Springer-Verlag.
- [38] Yuan Yu, Panagiotis Manolios, and Leslie Lamport. Model checking TLA+ specifications. In *CHARME '99: Proceedings of the 10th IFIP WG 10.5 Advanced Research Working Conference on Correct Hardware Design and Verification Methods*, pages 54–66, London, UK, 1999. Springer-Verlag.