Real-Time and Fault-Tolerant Systems
– Specification, verification, refinement and scheduling

Zhiming Liu and Mathai Joseph

May 2005

UNU/IIST Report No. 323

$\boxed{\mathcal{R}}$

Real-Time and Fault-Tolerant Systems
– Specification, verification, refinement and scheduling

Zhiming Liu and Mathai Joseph

Abstract

Fault-tolerance and timing have often been considered to be implementation issues of a program, quite distinct from the functional safety and liveness properties. Recent work has shown how these non-functional and functional properties can be verified in a similar way. However, the more practical question of determining whether a real-time program will meet its deadlines, i.e., showing that there is a feasible schedule, is usually done using scheduling theory, quite separately from the verification of other properties of the program. This makes it hard to use the results of scheduling analysis in the design, or redesign, of fault-tolerant and real-time programs. In this paper we show how fault-tolerance, timing, and schedulability can be specified and verified using a single notation and model. This allows a unified view to be taken of the functional and nonfunctional properties of programs and a simple transformational method to be used to combine these properties. It also permits results from scheduling theory to be interpreted and used within a formal proof framework. The notation and model are illustrated using a simple example.

Keywords: Specification, Verification, Refinement, Real-Time, Fault-Tolerance, Scheduling, TLA

Zhiming Liu is a research fellow at UNU/IIST. His research interests include theory of computing systems, including sound methods for specification, verification and refinement of fault-tolerant, real-time and concurrent systems, and formal techniques for OO development. His teaching interests are Communication and Concurrency, Concurrent and Distributed Programming, Internet Security, Software Engineering, Formal specification and Design of Computer Systems. E-mail: Z.Liu@iis.unu.edu.

Mathai Joseph is Executive Vice President of Tata Consultancy Services and the Executive Director of Tata Research and Design Development Centre (TRDDC), Pune, India. During 1986-1994, he worked as a professor in Computer Science at the University of Warwick in England. Professor. Mathai Joseph is the Chairman of the Board of UNU-IIST. In addition to his management position, Prof. Joseph is interested in applications of formal method to software tool development for industry software. E-Mail: mathai.joseph@tcs.com.

Contents

1 Introduction

A real-time system must meet functional and timing properties when implemented on a chosen hardware platform. Some timing properties can be derived from the specification of the system and others from the design choices made in the implementation. Yet other properties can be determined only by examining the timing characteristics of the implementation.

Real-time systems often need to meet critical safety requirements under a variety of operating conditions. One factor that then needs attention is the ability of the system to overcome the effects of faults that may occur in the system. Such faults are usually defined in terms of a fault-model. The degree of *fault-tolerance* of the system must be established in terms of the fault-model and the effect of faults upon the execution of the system.

In this chapter, we show that functional and many non-functional properties of a real-time system, such as schedulability, or proving that its implementation meets its timing constraints, can be verified in a similar way. Likewise, the fault-tolerance of a system can be proved using the same techniques. We use a single notation and model and take a unified view of the functional and non-functional properties of programs. A simple transformational method is used to combine these properties [58, 59]. We show how the theory of concurrency, fault-tolerance, real-time and scheduling can be built on the theories of sequential programming, such as those of Dijkstra's calculus of weakest preconditions [24], Hoare Logic [28], Morgan's refinement calculus [70] and Hoare and He's UTP [33]. These theories are discussed and used in Chapter 3 and Chapter 5.

Section 2 gives an informal account of real-time systems. Section 3 presents a historic background on formal techniques in real-time and fault-tolerance. Section 4 gives an outline of the approach used in this chapter. Section 5 introduces the computational model and the Temporal Logic of Actions [43] used for program specification, verification, and refinement. In Section 6, we show how physical faults are specified, how fault-tolerance is achieved by transforming a non-fault-tolerant program, and how fault-tolerance is verified and refined. Section 7 extends the method given in Section 5 for the specification and verification of real-time programs. In Section 8 we combine the techniques used for fault-tolerance and real-time. Section 9 shows how real-time scheduling policies can be specified and combined with the program specification for verification of schedulability of a program. Proof rules for feasibility and fault-tolerant feasibility are also developed and it is shown how methods and results from scheduling theory can be formally verified and used. The notation, model and techniques are illustrated using a simple processor-memory interface program.

2 Real-Time Systems: An Informal Account

Consider a car moving along a road that passes through some hills. Assume that there is an external observer who is recording the movement of the car using a pair of binoculars and a stopwatch. With a fast moving car, the observer must move the binoculars at sufficient speed to keep the car within sight. If the binoculars are moved too fast, the observer will view an area before the car has reached there; too slow, and the car will be out of sight because it is ahead of the viewed area. If the car changes speed or direction, the observer must adjust the movement of the binoculars to keep the car in view; if the car disappears behind a hill, the observer must use the cars recorded time, speed and direction to predict when and where it will re-emerge.

Suppose that the observer replaces the binoculars by an electronic camera which requires n seconds to process each frame and determine the position of the car. When the car is behind a hill, the observer must predict the position of the car and point the camera so that it keeps the car in the frame even though it is seen only at intervals of n seconds. To do this, the observer must model the movement of the car and, based on its past behaviour, predict its future movement. The observer may not have an explicit model of the car and may not even be conscious of doing the modelling; nevertheless, the accuracy of the prediction will depend on how faithfully the observer models the actual movement of the car.

Finally, assume that the car has no driver and is controlled by commands radioed by the observer. Being a physical system, the car will have some inertia and a reaction time, and the observer must use an even more precise model if the car is to be controlled successfully. Using information obtained every n seconds, the observer must send commands to adjust throttle settings and brake positions, and initiate changes of gear when needed. The difference between a driver in the car and the external observer, or remote controller, is that the driver has a continuous view of the terrain in front of the car and can adjust the controls continuously during its movement. The remote controller gets snapshots of the car every n seconds and must use these to plan changes of control.

2.1 Real-time computing

A real-time computer controlling a physical device or process has functions very similar to those of the observer controlling the car. Typically, sensors will provide readings at periodic intervals and the computer must respond by sending signals to actuators. There may be unexpected or irregular events and these must also receive a response. In all cases, there will be a time-bound within which the response should be delivered. The ability of the computer to meet these demands depends on its capacity to perform the necessary computations in the given time.

If a number of events occur close together, the computer will need to schedule the computations so that each response is provided within the required time-bounds. It may be that, even so, the system is unable to meet all the possible demands and in this case we say that the system lacks sufficient resources (since a system with unlimited resources and capable of processing at infinite speed could satisfy any such timing constraint). Failure to meet the timing constraint for a response can have different consequences: in some cases, there may be no effect at all; in other cases, the effects may be minor and correctable; in yet other cases, the results may be catastrophic. Looking at the behaviour required of the observer allows us to define some of the properties needed for successful real-time control.

A real-time program must

- interact with an environment which has time-varying properties,

- exhibit predictable time-dependent behaviour, and
- execute on a system with limited resources.

Let us compare this description with that of the observer and the car. The movement of the car through the terrain certainly has time-varying properties (as must any movement). The observer must control this movement using information gathered by the electronic camera; if the car is to be steered safely through the terrain, responses must be sent to the car in time to alter the setting of its controls correctly. During normal operation, the observer can compute the position of the car and send control signals to the car at regular intervals.

If the terrain contains hazardous conditions, such as a flooded road or icy patches, the car may behave unexpectedly, e.g. skidding across the road in an arbitrary direction. If the observer is required to control the car under all conditions, it must be possible to react in time to such unexpected occurrences. When this is not possible, we can conclude that the real-time demands placed on the observer, under some conditions, may make it impossible to react in time to control the car safely. In order for a real-time system to manifest predictable time-dependent behaviour it is thus necessary for the environment to make predictable demands. With a human observer, the ability to react in time can be the result of skill, training, experience or just luck. How do we assess the real-time demands placed on a computer system and determine whether they will be met? If there is just one task and a single processor computer, calculating the real-time processing load may not be very difficult. As the number of tasks increases, it becomes more difficult to make precise predictions; if there is more than one processor, it is once again more difficult to obtain a definite prediction. There may be a number of factors that make it difficult to predict the timing of responses [13].

- A task may take different times under different conditions. For example, predicting the speed of a vehicle when it is moving on level ground can be expected to take less time than if the terrain has a rough and irregular surface. If the system has many such tasks, the total load on the system at any time can be very difficult to calculate accurately.
- Tasks may have dependencies: Task A may need information from Task B before it can complete its calculation, and the time for completion of Task B may itself be variable. Under these conditions, it is only possible to set minimum and maximum bounds within which Task A will finish.
- With large and variable processing loads, it may be necessary to have more than one processor in the system. If tasks have dependencies, calculating task completion times on a multi-processor system is inherently more difficult than on a single processor system.
- The nature of the application may require distributed computing, with nodes connected by communication lines. The problem of finding completion times is then even more difficult, as communication between tasks can take varying times.

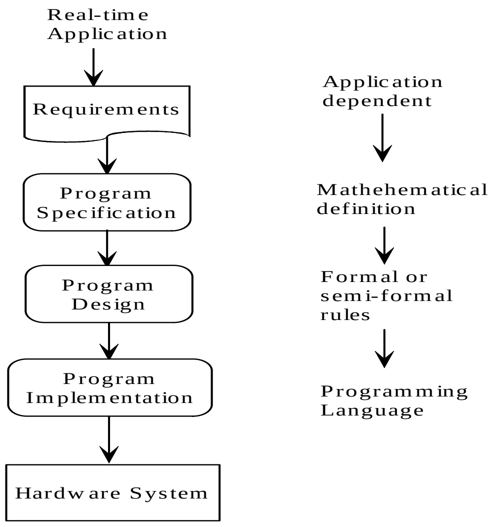

Figure 1: Real-Time System Development

Requirements, specification and implementation

The demands placed on a real-time system arise from the needs of the application and are often called the requirements. Deciding on the precise requirements is a skilled task and can be carried out only with very good knowledge and experience of the application. Failures of large systems are often due to errors in defining the requirements. For a safety related real-time system, the operational requirements must then go through a hazard and risk analysis to determine the safety requirements. Requirements are often divided into two classes: functional requirements, which define the operations of the system and their effects, and non-functional requirements, such as timing properties. A system which produces a correctly calculated response but fails to meet its timing-bounds can have as dangerous an effect as one which produces a spurious result on time. So, for a real-time system, the functional and non-functional requirements must be precisely defined and together used to construct the specification of the system.

A specification is a mathematical statement of the properties to be exhibited by a system. A specification should be abstract so that

– it can be checked for conformity against the requirement, and
– its properties can be examined independently of the way in which it will be implemented, i.e. as a program executing on a particular system.

This means that a specification should not enforce any decisions about the structure of the software, the programming language to be used or the kind of system on which the program is to be executed: these are properly implementation decisions. A specification is transformed into an application by taking design decisions, using formal or semi-formal rules, and converted into a program in some language (see Figure 1). We shall consider how a real-time system can be specified and implemented to meet the requirements. A notation will be used for the specification and it will be shown how the properties of the implementation can be checked. It will be noticed as the specifications unfold that there are many hidden complexities in even apparently simple real-time problems. This is why mathematical description and analysis have an important role to play, as they help to deal with this complexity. For both classical scheduling analysis and formal specification and verification in different notations, we refer the reader to [13].

2.2 An example real-time system: mine pump

We illustrate the problem of real-time by a well-known case study [13]. Water percolating into a mine is collected in a sump to be pumped out of the mine (see Figure 2). The water level sensors D and E detect when water is above a high and a low level respectively. A pump controller switches the pump on when the water reaches the high water level and off when it goes below the low water level. If, due to a failure of the pump, the water cannot be pumped out, the mine must be evacuated within one hour.

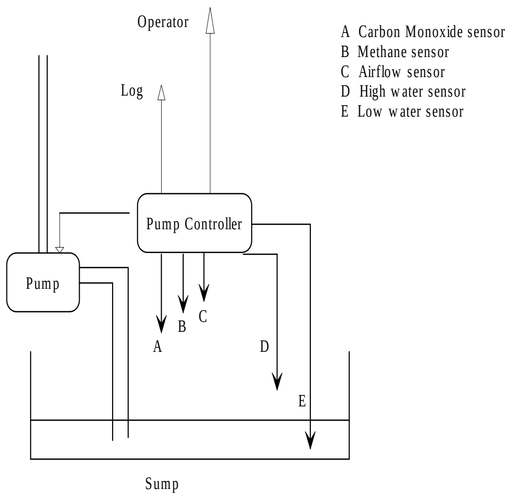

Figure 2: Mine pump and control system (originally from Burns and Lister, 1991)

The mine has other sensors (A, B, C) to monitor the carbon monoxide, methane and airflow levels. An alarm must be raised and the operator informed within one second of any of these levels becoming critical so that the mine can be evacuated within one hour. To avoid the risk of explosion, the pump must be operated only when the methane level is below a critical level.

Human operators can also control the operation of the pump, but within limits. An operator can switch the pump on or off if the water is between the low and high water levels. A special operator, the supervisor, can switch the pump on or off without this restriction. In all cases, the methane level must be below its critical level if the pump is to be operated.

Readings from all sensors, and a record of the operation of the pump, must be logged for later analysis.

Safety requirements

From the informal description of the mine pump and its operations we obtain the following safety requirements:

1. The pump must not be operated if the methane level is critical.
2. The mine must be evacuated within one hour of the pump failing.
3. Alarms must be raised if the methane level, the carbon monoxide level or the air-flow level is critical.

Operational requirement

The mine is normally operated for three shifts a day, and the objective is for no more than one shift in 1000 to be lost due to high water levels.

Problem Write and verify a specification for the mine pump controller under which it can be shown that the mine is operated whenever possible without violating the safety requirements.

Comments The specification is to be the conjunction of two conditions: the mine must be operated when possible, and the safety requirements must not be violated. If the specification read The mine must not be operated when the safety requirements are violated, then it could be trivially satisfied by not operating the mine at all! The specification must obviate this easy solution by requiring the mine to be operated when it is safely possible.

Note that the situation may not always be clearly defined and there may be times when it is difficult to determine whether operating the mine would violate the safety requirements. For example, the pump may fail when the water is at any level; does the time of one hour for the evacuation of the mine apply to all possible water levels? More crucially, how is pump failure detected? Is pump failure always complete or can a pump fail partially and be able to displace only part of its normal output?

It is also important to consider under what conditions such a specification will be valid. If the methane or carbon monoxide levels can rise at an arbitrarily fast rate, there may not be time to evacuate the mine, or to switch off the pump. Unless there are bounds on the rate of change of different conditions, it will not be possible for the mine to be operated and meet the safety requirements. Sensors operate by sampling at periodic intervals and the pump will take some time to start and to stop. So the rate of change of a level must be small enough for conditions to not become dangerous during the reaction time of the equipment.

The control system obtains information about the level of water from the *Highwater* and *LowWater* sensors and of methane from the *Methane* sensor. Detailed data is needed about the rate at which water can enter the mine, and the frequency and duration of methane leaks; the correctness of the control software is predicated on the accuracy of this information. Can it also be assumed that the sensors always work correctly?

The description explains conditions under which the mine must be evacuated but does not indicate how often this may occur or how normal operation is resumed after an evacuation. For example, can a mine be evacuated more than once in a shift? After an evacuation, is the shift considered to be lost? If the mine is evacuated, it would be normal for a safety procedure to come into effect and for automatic and manual clearance to be needed before operation of the mine can resume. This information will make it possible to decide on how and when an alarm is reset once it has been raised.

2.3 Developing a specification

We shall start by describing the requirements in terms of some properties, using a simple mathematical notation. This is a first step towards making a formal specification and we shall see various different, more complete, specifications of the problem in later chapters. Properties will be defined with simple predicate calculus expressions using the logical operators $\wedge$ (and), $\vee$ (or), $\Rightarrow$ (implies) and $\Leftrightarrow$ (iff), and the universal quantifier $\forall$ (for all). The usual mathematical relational operators will be used and functions, constants and variables will have types. We use

$$F : T_1 \rightarrow T_2$$

for a function F from type T_1 (the domain of the function) to type T_2 (the range of the function) and a variable V of type T will be defined as $V : T$. An interval from C_1 to C_2 will be represented as $[C_1, C_2]$ if the interval is closed and includes both C_1 and C_2, as $(C_1, C_2]$ if the interval is half-open and includes C_2 and not C_1 and as $[C_1, C_2)$ if the interval is half-open and includes C_1 and not C_2.

Assume that time is measured in seconds and recorded as a value in the set Time and the depth of the water is measured in metres and is a value in the set *Depth*; *Time* and *Depth* are the set of real numbers.

S1: Water level
The depth of the water in the sump depends on the rate at which water enters and leaves the sump and this will change over time. Let us define the water level *Water* at any time to be a function from *Time* to *Depth*:

$$\mathit{Water} : \mathit{Time} \rightarrow \mathit{Depth}$$

Let $Flow$ be the rate of change of the depth of water measured in metres per second and be represented by a real number; *WaterIn* and *WaterOut* are the rates at which water enters and leaves the sump and, since these rates can change, they are functions from *Time* to *Flow*:

$$\mathit{WaterIn},\ \mathit{WaterOut} : \mathit{Time} \rightarrow \mathit{Flow}$$

The depth of water in the sump at time t_2 is the sum of the depth of water at an earlier time t_1 and the difference between the amount of water that flows in and out in the time interval $[t1, t2]$. Thus $\forall t1, t2 : Time, t1 \leq t2$;

$$\mathit{Water}(t_2) = \mathit{Water}(t_1) + \int_{t_1}^{t_2} (\mathit{WaterIn}(t) - \mathit{WaterOut}(t))dt$$

HighWater and *LowWater* are constants representing the positions of the high and low water level sensors. For safe operation, the pump should be switched on when the water reaches the level *HighWater* and the level of water should always be kept below the level *DangerWater*:

$$\mathit{DangerWater} > \mathit{HighWater} > \mathit{LowWater}$$

If $\mathit{HighWater} = \mathit{LowWater}$, the high and low water sensors would effectively be reduced to one sensor.

S2: Methane level
The presence of methane is measured in units of pascals and recorded as a value of type *Pressure* (a real number). There is a critical level, *DangerMethane*, above which the presence of methane is dangerous.

The methane level is related to the flow of methane in and out of the mine. As for the water level, we define a function *Methane* for the methane level at any time and the functions *MethaneIn* and $MethaneOut$ for the flow of methane in and out of the mine:

$$\begin{array}{l}\mathit{Methane} : \mathit{Time} \Rightarrow \mathit{Pressure} \\ \mathit{MethaneIn}, \mathit{MethaneOut} : \mathit{Time} \Rightarrow \mathit{Pressure}\end{array}$$

and $\forall t_1, t_2 : Time$,

$$\mathit{Methane}(t_2) = \mathit{Methane}(t_1) + \int_{t_1}^{t_2} (\mathit{MethaneIn}(t) - \mathit{MethaneOut}(t))dt$$

S3: Assumptions

1. There is a maximum rate $\mathit{MaxWaterIn} : \mathit{Flow}$ at which the water level in the sump can increase and at any time t, $\mathit{WaterIn}(t) \leq \mathit{MaxWaterIn}$.
2. The pump can remove water with a rate of at least $\mathit{PumpRate} : \mathit{Flow}$, and this must be greater than the maximum rate at which water can build up: $\mathit{MaxWaterIn} < \mathit{PumpRate}$.
3. The operation of the pump is represented by a predicate on Time which indicates when the pump is operating:

   $$\mathit{Pumping} : \mathit{Time} \rightarrow \mathit{Bool}$$

   and if the pump is operating at any time t it will produce an outflow of water of at least *PumpRate*:

   $$(\mathit{Pumping}(t) \wedge \mathit{Water}(t) > 0) \Rightarrow (\mathit{WaterOut}(t) > \mathit{PumpRate})$$

4. There is enough reaction time t_P before the water level becomes dangerous;

   $$(\mathit{HighWater} + \mathit{MaxWaterIn} \cdot (t_P)) < \mathit{DangerWater}$$

5. The maximum rate at which methane can enter the mine is *MaxMethaneRate*.

   If the methane sensor measures the methane level periodically every t_M units of time, and if the time for the pump to switch on or off is t_P, then the reaction time $t_M + t_P$ must be such that,

$$\begin{array}{l}(\mathit{MaxMethaneRate} \cdot t_m + \mathit{HighMethane}) < \mathit{MethaneMargin} \wedge \\ (\mathit{MaxMethaneRate} \cdot t_P + \mathit{MethaneMargin}) < \mathit{DangerMethane}\end{array}$$

   where *HighMethane* $<$ *MethaneMargin* $<$ *DangerMethane*. *HighMethane* is the safety limit of methane and the methane is below this limit when the system starts. The controller should start to turn the pump off when it receives a methane level greater than *HighMethane* signal from the sensor.

6. The methane level does not reach *MethaneMargin* more than once in 1000 shifts; without this limit, it is not possible to meet the operational requirement. Methane is generated naturally during mining and is removed by ensuring a sufficient flow of fresh air, so this limit has some implications for the air circulation system.

S4: Pump controller

The pump controller must ensure that, under the assumptions, the operation of the pump will keep the water level within limits. At all times when the water level is high and the methane level is not critical, the pump is switched on, and if the methane level is critical the pump is switched off. Ignoring the reaction times, this can be specified as follows:

$$\forall t \in \mathit{Time} \cdot \left(\begin{array}{l} \mathit{Water}(t) > \mathit{HighWater} \wedge \\ \mathit{Methane}(t) < \mathit{DangerMethane} \end{array} \right) \Rightarrow \mathit{Pumping}(t)$$

$$\wedge (\mathit{Methane}(t) \geq \mathit{DangerMethane}) \Rightarrow \neg \mathit{Pumping}(t))$$

This cannot really be achieved so let us see how reaction times can be taken into account. Since t_P is the time taken to switch the pump on, a properly operating controller must ensure that:

$$\forall t \in \mathit{Time} \cdot \left(\begin{array}{l} \mathit{Methane}(t) < \mathit{HighMethane} \wedge \neg \mathit{Pumping}(t) \wedge \\ \mathit{Water}(t) \geq \mathit{HighWater} \end{array} \right. \\ \Rightarrow \exists t_0 \leq t_P \cdot \mathit{Pumping}(t + t_0) \left. \vphantom{\begin{array}{l}a\\b\end{array}} \right)$$

So if the operator has not already switched the pump on, the pump controller must do so when the water level reaches *HighWater*. Similarly, the methane sensor may take t_M units of time to detect a methane level and the pump controller must ensure that

$$\forall t \in Time \cdot \left(\begin{array}{l} \mathit{Pumping}(t) \wedge \\ \mathit{Methane}(t) \geq \mathit{HighMethane} \end{array} \right) \Rightarrow \exists t_0 \leq t_p \cdot \neg \mathit{Pumping}(t + t_0)$$

S5: Sensors

Sensors are modelled by variables. The high water sensor provides information about the height of the water at time t in the form of predicates $HW(t)$ and $LW(t)$ which represent the cases where the water level is above *HighWater* and $LowWater$ respectively. We assume that at all times a correctly working sensor gives some reading (i.e. $HW(t) \vee \neg HW(t)$).

The readings provided by the sensors are related to the actual water level in the sump:

$$\begin{aligned}\forall t \in Time\cdot\quad & \textit{Water}(t) \geq \textit{HighWater} \Rightarrow HW(t)\\ & \wedge \textit{Water}(t) \geq \textit{LowWater} \Rightarrow LW(t)\end{aligned}$$

Similarly, the methane level sensor reads the methane level periodically and signals to the controller that either $HML(t)$ or $\neg HML(t)$:

$$\begin{aligned}\forall t \in \textit{Time}\cdot\quad & \textit{Methane}(t) \geq \textit{HighMethane} \Rightarrow \textit{HML}(t)\\ & \wedge \textit{Methane}(t) < \textit{HighMethane} \Rightarrow \neg \textit{HML}(t)\end{aligned}$$

S6: Actuators
The pump is switched on and off by an actuator which receives signals from the pump controller. Once these signals are sent, the pump controller assumes that the pump acts accordingly. To validate this assumption, another condition is set by the operation of the pump. The outflow of water from the pump sets the condition $PumpOn$; similarly, when there is no outflow, the condition is *PumpOff*.

The assumption that the pump really is pumping when it is on and is not pumping when it is off is specified below: assume the pump takes κ time units to react:

$$\begin{aligned}\forall t \in \textit{Time}\cdot\quad & \textit{PumpOn}(t) \Rightarrow \exists t_0 \leq \kappa \cdot \textit{Pumping}(t + t_0)\\ & \textit{PumpOff}(t) \Rightarrow \exists t_0 \leq \kappa \cdot \neg \textit{Pumping}(t + t_0)\end{aligned}$$

We can then refine the specification of the controller as

$$\begin{aligned}& HW(t) \wedge LM(t) \Rightarrow \exists t_o \leq \varepsilon \cdot \textit{PumpOn}(t + t_o)\\ & HW(t) \wedge HM(t) \Rightarrow \exists t_o \leq \varepsilon \cdot \textit{PumpOff}(t + t_o)\end{aligned}$$

where $\varepsilon + \kappa \leq t_P$.

The condition *PumpOn* is set by the actual outflow and there may be a delay before the outflow changes when the pump is switched on or off. If there were no delay, the implication $\Rightarrow$ could be replaced by the two-way implication iff, represented by $\Leftrightarrow$, and the two conditions *PumpOn* and *PumpOff* could be replaced by a single condition.

The *verification* of the system specification is about to prove

$$\begin{aligned}& (\text{Controller Specification}) \wedge (\text{Actuator Specification}) \wedge (\text{Sensors Sepecification}) \Rightarrow\\ & (\text{Assumptions} \Rightarrow (\text{Requirement Specification}))\end{aligned}$$

2.4 Constructing the specification

The simple mathematical notation used so far provides a more abstract and a more precise description of the requirements than does the textual description. Having come so far, the next step should be to combine the definitions given in **S1 – S6** and use this to prove the safety properties of the system. The combined definition should also be suitable for transformation into a program specification which can be used to develop a program.

Unfortunately, this is where the simplicity of the notation is a limitation. The definitions **S1 – S6** can of course be made more detailed and perhaps taken a little further towards what could be a program specification. But the mathematical set theory used for the specification is both too rich and too complex to be useful in supporting program development. To develop a program, we need to consider several levels of specification (and so far we have just outlined the beginnings of one level) and each level must be shown to preserve the properties of the previous levels. The later levels must lead directly to a program and an implementation and there is nothing so far in the notation to suggest how this can be done.

What we need is a specification notation that has an underlying computational model which holds for all levels of specification. The notation must have a calculus or a proof system for reasoning about specifications and a method for transforming specifications to programs.

2.5 Analysis and implementation

The development of a real-time program takes us part of the way towards an implementation. The next step is to analyze the timing properties of the program and, given the timing characteristics of the hardware system, to show that the implementation of the program will meet the timing constraints. It is not difficult to understand that for most time-critical systems, the speed of the processor is of great importance. But how exactly is processing speed related to the statements of the program and to timing deadlines?

A real-time system will usually have to meet many demands within limited time. The importance of the demands may vary with their nature (e.g. a safety-related demand may be more important than a simple data-logging demand) or with the time available for a response. The allocation of the resources of the system needs to be planned so that all demands are met by the time of their deadlines. This is usually done using a scheduler which implements a scheduling policy that determines how the resources of the system are allocated to the program. Scheduling policies can be analyzed mathematically so the precision of the formal specification and program development stages can be complemented by a mathematical timing analysis of the program properties. Taken together, specification, verification and timing analysis can provide accurate timing predictions for a real-time system.

We will discuss the relation between schedulability and verification and refinement.

3 Historical Background of Formal Techniques in Real-Time and Fault-Tolerance

Starting from the early work in the 1970's, formal methods for concurrent and distributed systems development have seen considerable development. They have made a significant contribution to a better understanding of the behaviour of concurrent and distributed systems and to their correct and reliable implementation. The most widely studied methods include:

– Transition systems with temporal logic [74, 65, 30].
– Automata with Temporal logic [20, 11].
– Process algebras [32, 68].

Traditional temporal logic methods (and similar formalisms) use a *discrete event* approach: this is also the case with *transition systems* (e.g. [39, 74, 64, 41, 75]), *automata* (e.g. [20, 21]) and *action systems* (e.g. [9, 17]). Such models abstract away time in the behaviour and describe the ordering of the events of a system.

Real-time is introduced into transition systems either by associating lower and upper bounds with enabled transitions [76] or by introducing explicit clocks [6, 3]. For specification and verification, a temporal logic is then extended either with the introduction of bounded (or *quantized*) temporal operators [76, 40] or with the addition of explicit clock variables [76, 5, 3]. The relationship between the two approaches, the extent to which one can be translated into another, is investigated in [30].

One approach to the construction of safe and dependable computing systems is to use formal specification, development and verification methods as part of a *fault-intolerance approach* in which the system safety and dependability are improved by *a priori fault avoidance* and *fault removal* [8]. Another path towards this goal is through *fault-tolerance*, which is complementary to fault-intolerance but not a substitute for it [8]. This is based on the use of protective redundancy: a system is designed to be *fault-tolerant* by incorporating additional components and algorithms to ensure that the occurrence of an error state does not result in later system failures [77, 78, 46]. Although fault-tolerance is by no means a new concept [71], there was little work on formal treatment of fault-tolerance until the 1980s [69, 81, 23, 50, 53, 54, 55]. These papers treat untimed fault-tolerant systems only. Recent work [22, 80, 45, 56] has shown how fault-tolerance and timing properties can formally be treated uniformly.

The issue of schedulability arises when a real-time program is to be implemented on a system with limited resources (such as processors) [37]. An infeasible implementation of a real-time program will not meet the timing requirement even though the program has been formally proven correct. Schedulability has been for a long time a concern of scheduling theory (e.g. [49, 38, 47, 7, 14]) but the models and techniques used there are quite different from those used in formal specification and development methods. The relationship between the computational model used in a scheduling analysis and the model (e.g. an interleaving model) used in a formal development is not clear. Thus, results obtained in scheduling theory are hard to relate to or use in the formal development of a system. It is however possible to verify the schedulability of a program within a formal framework [73, 85, 27, 60, 63] and this provides a starting point for a proof-theoretic interpretation of results from scheduling theory.

4 Overview of the Formal Framework

We now show how fault-tolerance and schedulability, as well as functional and time correctness, can be specified and verified within a single formal framework. We use *transition systems* [39, 74] as the program model, and the Temporal Logic of Actions (TLA) [43, 44] as the specification notation. Physical faults in a system are modelled as being caused by a set F of 'fault actions' which perform state transformations in the same way as other program actions. Fault-tolerance is achieved if the program can be made tolerant to these faults (e.g. by adding the appropriate recovery actions [77, 78, 50, 53, 54, 55]). We shall show that proof of fault-tolerance is no different to proof of any functional property.

Each action τ of a real-time program is associated with a *volatile lower bound* $L(\tau)$ and a *volatile upper bound* $U(\tau)$, meaning that 'action τ can be performed only if it has been *continuously* enabled for at least $L(\tau)$ time units, and τ must not be *continuously* enabled for $U(\tau)$ time units without being performed'. The use of volatile time bounds or, correspondingly, *volatile timers* (or *clock variables*) in the explicit-clock modelling approach has been described in the literature (e.g. see the references in the previous subsection) to specify the time-criticality of an operation.

To deal with real-time scheduling, it is important to model actions and their pre-emption at a level of abstraction suitable for measuring time intervals and to ensure that pre-emption of an execution respects the atomicity of actions. To achieve this, we use *persistent time bounds* [57] to constrain the *cumulative execution time* of an action in the execution of a program under a scheduler. The persistent lower bound $l(\tau)$ for an action τ means that 'action τ can be performed (or finished) only if it has been executed by a processor for at least a *total* of $l(\tau)$ time units, not necessarily continuously'; the persistent upper bound $u(\tau)$ means that 'τ is not executed by a processor for a *total* of $u(\tau)$ time units without being completed'.

In TLA, programs and properties are specified as logical formulas, and this allows the logical characterisation and treatment of the *refinement relation* between programs. We shall show how, using this approach, the untimed program, the fault assumptions, the timing assumptions, and scheduling policies are specified as separate TLA formulas.

The use of a well established computational model and logic has significant advantages over the use of a specially designed semantic model and logic (e.g. as in [81, 23, 22, 80] for fault-tolerance and [73, 63, 26] for schedulability). First, less effort is needed to understand the model and the logic. Second, existing methods for specification, refinement and verification can be readily applied to deal with fault-tolerance and schedulability. Also, existing mechanical proof assistance (e.g. [25, 10] and model-checking methods and tools (e.g. [20, 4, 31]) can be used [6, 5, 11].

5 Program Specification, Verification and Refinement

This section introduces a transition system which is widely used as the computational model in temporal logic. This model serves as the semantic model of TLA that we shall use for specification and verification.

5.1 Introducing TLA

Values, variables and states TLA is a logic used for specifying and reasoning about programs which manipulate data. Assume there is a set *Val* of *values*, where a value is a data item. We assume that *Val* contains all the values, such as numbers like 3, strings such as "abc" and sets like *Nat*, needed for our programs.

Assume that a program manipulates data by changing its *state*, which is an assignment of values to *state variables*. For describing all possible programs, we assume an *infinite* set *Var* of variables, which are represented by symbols like x, y, z. A *state* s is thus a mapping from *Var* to *Val*:

$$s : \mathit{Var} \longmapsto \mathit{Val}$$

For a state s, the value assigned to a variable x in state s is represented by $s[x]$ and the values assigned to a subset $\overline{z}$ of variables is denoted by $s[\overline{z}]$. Given a subset $\overline{v} \subseteq \mathit{Val}$ of variables, we also define *a state* s *over* $\overline{v}$ to be a mapping from $\overline{v}$ to *Val*.

5.1.1 Examples of states

For state variables: $\{x, y\}$, let

- $s = \{x \mapsto 0, y \mapsto 1\}$, $s' = \{x \mapsto 1, y \mapsto 0\}$
- $s[x] = 0$, $s[y] = 1$, $s'[x] = 1$, $s'[y] = 0$

Assume $\{\mathit{On}, \mathit{Off}, \mathit{Bright}\}$ are state variables used to model a light system:

- $s_1 = \{\mathit{Off} \mapsto true, \mathit{On} \mapsto false, \mathit{Bright} \mapsto false\}$
  $s_2 = \{\mathit{Off} \mapsto false, \mathit{On} \mapsto true, \mathit{Bright} \mapsto false\}$
  $s_3 = \{\mathit{Off} \mapsto false, \mathit{On} \mapsto true, \mathit{Bright} \mapsto true\}$
- $s_1[\mathit{On}] = false$, $s_2[\mathit{On}] = true$, etc.

♣

State predicates A *state predicate*, called a *predicate* for short, is a first-order Boolean-valued expression built from variables and constant symbols. For example, $(x = y - 3) \wedge x \in Nat$. The meaning $[\![Q]\!]$ of a predicate is a mapping from states to Booleans $\{true, false\}$ once an *interpretation* is given to the predicate symbols like "$=$" and the function symbols like "$-$" used in Q. We say that a state s *satisfies* a predicate Q, denoted by $s \models Q$, iff $[\![Q]\!](s) = true$.

Consider variables $\{x, y\}$ and the states $s = \{x \mapsto 0, y \mapsto 1\}$ and $s' = \{x \mapsto 1, y \mapsto 0\}$. Then $(x - 1 = y)$, $(x + y > 3)$, and $(x - 1 = y) \vee (x + y > 3)$ are all predicates. Assume x and y take values from the integers and the meanings of the equality symbol "$=$", inequality symbol "$>$", and the function symbols "$-$" and "$+$" are those defined in the arithmetic on integers. We can easily decide which of the predicates are satisfied by state s and which by s'.

Actions The execution of a program changes the state of the program by the execution of *atomic actions*, called an *actions* for short. An *action* is a first-order Boolean-valued expression over the variables *Var* and their 'primed versions' *Var*$'$. For example, $x' + 1 = y$ and $x' \geq y' + (x - 1)$ are actions.

For a given interpretation of the predicate symbols such as "$=$" and "$\geq$" and an interpretation of the function symbols such as "$+$" and "$-$", an action defines a relation between the *values* of variables before and the *values* of primed variables after the execution of the action. Formally, given the interpretation of the predicate and function symbols, the meaning $[\![\tau]\!]$ of an action τ is a relation between states, i.e. a function that assigns a Boolean value to a pair (s, s') of states. We thus define $[\![\tau]\!](s, s')$ by considering s to be the *pre-τ-state* and s' the *post-τ-state* and $[\![\tau]\!](s, s')$ is obtained from τ by replacing each unprimed variable x it τ by its value $s[x]$ in s and each primed variable x' in τ by the value $s'[x]$ of x in s:

$$[\![\tau]\!](s, s') = true \text{ iff } \tau(s[\overline{z}]/\overline{z}, s'[\overline{z}]/\overline{z}') \text{ holds}$$

where $\overline{z}$ and $\overline{z}'$ are the sets of unprimed and primed variables in τ. We say that a pair (s, s') of states *satisfies* an action τ, denoted by $(s, s') \models \tau$, iff $[\![\tau]\!](s, s') = true$. When $(s, s') \models \tau$, (s, s') is called a *τ-step*.

A predicate Q can also be viewed as a particular action which does not have primed variables. Thus Q is satisfied by a pair (s, s') of states iff it is satisfied by the first state s in the pair. For an action τ, let *en*(τ) be the predicate, called the *enabling condition* (or *guard*) of τ, which is true of a state s iff there exists a state s' such that $(s, s') \models \tau$. Formally, let $x'_1, \ldots, x'_n$ be the primed variables that occur in τ, let $\hat{x}_1, \ldots, \hat{x}_n$ be new logical variables that do not occur in τ, and let $\hat{\tau}$ be the formula obtained from τ by replacing each occurrence of x'_i by $\hat{x}_i$, for $i = 1, \ldots, n$:

$$en(\tau) \triangleq \exists \hat{x}_1, \ldots \hat{x}_n . \hat{\tau}$$

Temporal formulas We consider an execution of a program to be an *infinite* state sequence, and take the semantics of the program to be the set of all its possible executions. Reasoning about programs is reasoning about their executions and thus reasoning about state sequences. We shall use TLA for this purpose.

Formulas in TLA are called *temporal formulas* which are built from actions as the *elementary temporal formulas* using Boolean connectives and modal operators in Linear-Time Temporal Logic [65]. Here we use only $\Box$ (read *always*) and its dual operator $\Diamond$ (read *eventually*) defined as $\neg\Box\neg$. Quantification (i.e. $\exists\overline{x}$, $\forall\overline{x}$) is possible over a set of *logical* (or *rigid*) variables, whose values are fixed over states, and over a set of state variables, whose values can change from state to state[1].

To use these formulas for describing state sequences, it requires to define the semantic meaning of such a formula as a function from executions to Booleans. We must first lift the semantics of an action based on pairs of states to one based on state sequences.

Given an infinite sequence $\sigma = \sigma_0, \sigma_1, \ldots$ of states,

- An action $[\![\tau]\!](\sigma) = true$ iff $[\![\tau]\!](\sigma_0, \sigma_1) = true$. Note that $[\![\tau]\!]$ is overloaded here.
- The first-order connectives and quantification over logical variables retain their standard semantics.
- $[\![\Box\varphi]\!](\sigma) = true$ iff $[\![\varphi]\!](\eta) = true$ for any suffix η of σ. This implies that $[\![\Diamond\varphi]\!](\sigma) = true$ iff $[\![\varphi]\!](\eta) = true$ for some suffix η of σ.
- $[\![\exists x.\varphi]\!](\sigma) = true$ iff there is an η such that $\sigma =_x \eta$ and $[\![\varphi]\!](\eta)$, where the relation $\sigma =_x \eta$ holds between state sequences σ and η iff $\sigma_i[y] = \eta_i[y]$ for any variable y which differs from x and for any $i \geq 0$. Thus, $\exists x.\varphi$ is true of σ iff φ is true of some infinite state sequence η that differs from σ only in the values assigned to the variable x.

We say that a formula φ is satisfied by σ, denoted by $\sigma \models \varphi$, if $[\![\varphi]\!](\sigma)$.

[1]In [43, 44], the bold versions $\exists$ and $\forall$ are used to quantify state variables.

5.1.2 Question

For state variables $\{x, y\}$ and a state sequence:

$$\sigma = s_1, s_2, s_1, s_2, \ldots$$

where $s_1 = \{x \mapsto 0, y \mapsto 1\}$ and $s_2 = \{x \mapsto 1, y \mapsto 0\}$, which of the following relations hold?

1. $\sigma \models (x = y - 1)$
2. $\sigma \models \Box(x + 1 = 1)$
3. $\sigma \models \Box(y + y' = 1)$
4. $\sigma \models \exists x.\Box((y \geq 0) \wedge (x = 1))$
5. $\sigma \models \Box\Diamond(y' = y - 1)$,
   $\sigma \models \Box\Diamond(y' = y + 1)$,
6. $\sigma \models \Box(\Diamond(y' = y - 1) \wedge \Diamond(y' = y + 1))$

♣

A formula φ is *valid* if it is satisfied by any infinite state sequences over *Var*. A relatively complete proof system is given in [43], with additional rules for using the logic for reasoning about programs. Every valid TLA formula is provable from the axioms and proof rules of the TLA proof system if all the valid action formulas are provable. As the temporal operators $\Box$ and $\Diamond$ and the semantic model are the same as those in [65], the rules and methods provided there for verification can also be used.

5.1.3 Question

Which of the following formulas are valid?

- $\Box(\varphi \vee \neg\varphi)$, $\Box\varphi \Rightarrow \varphi$, $\varphi \Rightarrow \Diamond\varphi$, $\Box\varphi \Rightarrow \Diamond\varphi$, $\Diamond(\varphi \vee \psi) \Rightarrow \Diamond\varphi \vee \Diamond\psi$
- $\Diamond(\varphi \wedge \psi) \Rightarrow \Diamond\varphi \wedge \Diamond\psi$, $\Box(\varphi \vee \psi) \Rightarrow \Box\varphi \vee \Box\psi$, $\Box(\varphi \wedge \psi) \Rightarrow \Box\varphi \wedge \Box\psi$

♣

5.2 The computational model and program specification

We now give a mathematical definition of a program.

Definition 1 *A* **program** *will be represented as an* action system *(or* a transition system*) which is a tuple $P = (\overline{v}, \overline{x}, \Theta, A)$ consisting of four components:*

1. *A finite non-empty set $\overline{v}$ of* state variables*.*

2. *A set $\overline{x}$ of* internal variables, *which is a subset of $\overline{v}$ and possibly empty. The values of these internal variables are not observable to the environment of the program.*
3. *An* initial condition Θ *which is a state predicate referring to only variables in $\overline{v}$ that defines the set of* initial states *of the program.*
4. *A finite set A of* atomic actions *in which only variables in $\overline{v}$ and primed variables in $\overline{v}'$ can occur..*

The simple light control system can also be modelled as

– $\overline{v} \triangleq \{s\}, \overline{x} \triangleq \{\ \}$
– $\Theta \triangleq (s = \mathit{off}$
– $A \triangleq \left\{ \begin{array}{ll} a : (s = \mathit{off}) \wedge (s' = \mathit{on}), & b : (s = \mathit{on}) \wedge (s' = \mathit{off}), \\ c : (s = \mathit{on}) \wedge (s' = \mathit{bright}), & d : (s = \mathit{bright}) \wedge (s' = \mathit{off}) \end{array} \right\}$

Notice that this model is not quite precise yet as it does not say what cannot be changed by an action. One can imagine think of combining action b and c by a disjunction $\vee$, that will illustrate the nondeterminism of a system.

Consider the composition of the two open systems:

– Light control system: *LightC*:

$$\begin{array}{l} (s = \mathit{off} \wedge \mathit{button} = \mathit{pressed}) \wedge (s' = \mathit{on} \wedge \mathit{button}' = \mathit{released}), \\ (s = \mathit{on} \wedge \mathit{button} = \mathit{pressed}) \wedge (s' = \mathit{off} \wedge \mathit{button}' = \mathit{released}), \\ (s = \mathit{on} \wedge \mathit{button} = \mathit{pressed}) \wedge (s' = \mathit{bright} \wedge \mathit{button}' = \mathit{released}), \\ (s = \mathit{bright} \wedge \mathit{button} = \mathit{pressed}) \wedge (s' = \mathit{off} \wedge \mathit{button}' = \mathit{released}) \end{array}$$

– The Button: *Button* with one action:

$$(\mathit{button} = \mathit{released}) \wedge (\mathit{button}' = \mathit{pressed})$$

$LightC \parallel Button$ has the union of the set of variables and the union of the sets of actions of the two components.

Definition 2 *A* **computation** *(an execution, a run) of program $P = (\overline{v}, \overline{x}, \Theta, A)$ is an infinite sequence $\sigma = \sigma_0, \sigma_1, \ldots$ over $\overline{v}$ such that the following two conditions hold:*

Initiality*:* σ_0 *satisfies* Θ*.*

Consecution*: For all $i \geq 0$, either $\sigma_i = \sigma_{i+1}$ (a* stuttering step*) or there is an action τ in A such that (σ_i, σ_{i+1}) is a τ-step (a* diligent step*). In the latter case, we say that a τ step is* taken *at position i of σ.*

Thus a computation either contains infinitely many diligent steps, or a diligent step takes it to a terminating state after which only stuttering steps occur; in this case we say that the computation is *terminating*.

The set of all the computations of a program is *stuttering closed*: if an infinite state sequence σ is a computation of the program, then so is any state sequence obtained from σ by adding or deleting a finite number of stuttering steps.

Remarks on atomicity, interleaving and concurrency

- Atomicity is a means of modelling *mutual exclusion synchronization*.
- Guarded atomic actions are for *conditional synchronization*. Notice that the guard of an action here is the enabling condition of the action.
- A number of atomic actions can be executed *in parallel iff* the order in which they take place does not affect the changes in the states. In this model, the use of an interleaving semantics works well for concurrent systems.
- An atomic action can be understood, and in fact *often though not always* implemented, as a piece of a sequential *terminating* program.
- A piece of a terminating concurrent program (nested parallelism) is equivalent to a non-deterministic sequential program (this ia also captured by the expansion law of CCS [67] and CSP [32]).

Notice that in this definition, an atomic action in a program is semantically taken just as a binary relation on the states. Therefore, although the actions in the set A are syntactically distinct from each other, we do not require that the actions be mutually disjoint in their semantics; in particular, one action can semantically be a sub-relation of another. This implies that it is possible that two actions have the same effect on a single state. This does not cause any theoretical problem, as we are to reason about properties of the execution of the program, not the effect of a individual action. In practice, when we use this model to define the semantics of a concurrent program, each atomic action defines a different piece of code of the program. Then the effect of all the actions obtained from the program will be different in any state, as they at least modify different control variables, such as process counter variables, which are usually internal variables.

An atomic action of a program usually changes only a subset of the variables of the program, leaving the others unchanged. For a finite set $\overline{z}$ of variables, we define:

$$\mathit{unchanged}(\overline{z}) = \bigwedge_{x \in \overline{z}} (x' = x)$$

For example, the atomic action in the form of the guarded command $x > 0 \longrightarrow x := x - 1$ can be described as the action formula:

$$(x > 0) \wedge (x' = x - 1) \wedge \mathit{unchanged}(\overline{v} - \{x\})$$

in which $x > 0$ is the enabling condition (i.e. the guard).

In the examples, we will simply omit the *unchanged* part when we specify an action, by assuming it changes the values of only those variables whose primed versions are referred to in the action formula.

To specify stuttering, we define also an abbreviation for an action τ and a finite set of state variables $\overline{z}$:

$$[\tau]_{\overline{z}} \triangleq \tau \vee \mathit{unchanged}(\overline{z})$$

asserting that a step is either a τ-step or a step which does not change the values of the state variables $\overline{z}$.

We are ready to define two normal forms of program specifications.

Definition 3 *Given a program $P = (\overline{v}, \overline{x}, \Theta, A)$, let:*

$$\mathcal{N}_P \triangleq \bigvee_{\tau \in A} \tau$$

$\mathcal{N}_P$ is the state-transition relation *for the atomic actions of P. The* **exact** *(or* **internal***)* specification *of P is expressed by the formula:*

$$\Pi(P) \triangleq \Theta \wedge \Box[\mathcal{N}_P]_{\overline{v}}$$

An exact specification defines all the possible sequences of values that may be taken by the state variables, including the *internal* variables $\overline{x}$. Existential quantification can be used to hide the internal variables $\overline{x}$ which 'automatically' get their adequate values although they are not visible to the observer.

Definition 4 *The* **canonical** *(or* **external***)* safety specification *of P is given as:*

$$\Phi(P) \triangleq \exists \overline{x}.\Pi(P)$$

An infinite state sequence σ over $\overline{v}$ satisfies $\Phi(P)$ iff there is an infinite state sequence η that satisfies $\Pi(P)$ and differs from σ only in the values assigned to the variables x_i, $i = 1, \dots, n$.

Importance of the stuttering closure property Stuttering closure is important when using a specification of a system in a larger system. In that case, the actions of this subsystem will be interleaved with other actions in the larger system, and the variables of the subsystem will not be changed when actions of the rest of the system take place.

To understand this point, consider a digital clock that displays only the hour. Let *hr* represents the clock's display.

From any starting hour, say 11, the behaviour of the clock is trivially:

$$\{hr \mapsto 11\} \longrightarrow \{hr \mapsto 12\} \longrightarrow \{hr \mapsto 1\} \longrightarrow \{hr \mapsto 2\} \cdots$$

Each step is carried out by the action

$$\textit{HCnext} \triangleq hr' = (hr \bmod 12) + 1$$

The clock can be specified by *HCinit* $\wedge$ $\Box$*HCnext*, where *HCinit* is the initial condition that the clock starts from any hour:

$$\textit{HCinit} \triangleq hr \in \mathbf{N} \wedge (1 \leq hr \leq 12)$$

This will work if the clock is considered in isolation and never related to another system. However, this specification cannot be re-used when we model a device that displays the current hour and temperature.

$$\left\{\begin{array}{l} hr \mapsto 11, \\ temp \mapsto 3.5 \end{array}\right\} \longrightarrow \left\{\begin{array}{l} hr \mapsto 12, \\ temp \mapsto 3.5 \end{array}\right\} \longrightarrow \left\{\begin{array}{l} hr \mapsto 12, \\ temp \mapsto 3 \end{array}\right\} \longrightarrow \left\{\begin{array}{l} hr \mapsto 12, \\ temp \mapsto 2.5 \end{array}\right\} \cdots$$

Therefore, stuttering is essential for composition. We will also see later that the stuttering closure property is the key to deal with refinement between two programs.

Exercise: Write a specification *Tempclock* for a digital clock that displays the current hour and temperature such that:

$$\mathit{TempClock} = \mathit{HClock} \wedge \mathit{TempDisplay}$$

where

$$\mathit{HClock} \triangleq \mathit{HCinit} \wedge [\mathit{HCnext}]_{hr}$$

♣

Formulas $\Pi(P)$ and $\Phi(P)$ are safety properties, i.e. they are satisfied by an infinite state sequence iff they are satisfied by every finite prefix of the sequence. Safety properties allow computations in which a system performs correctly for a while and then leaves the values of all variables unchanged.

For an action τ, define the action $\langle\tau\rangle_{\overline{z}} \triangleq \tau \wedge \neg \mathit{unchanged}(\overline{z})$. Then we can specify the following fairness properties.

$$\begin{array}{ll} \textit{Weak fairness:} & WF_{\overline{z}}(\tau) \triangleq (\Box\Diamond\langle\tau\rangle_{\overline{z}}) \vee (\Box\Diamond\neg \mathit{en}(\langle\tau\rangle_{\overline{z}})) \\ \textit{Strong fairness:} & SF_{\overline{z}}(\tau) \triangleq (\Box\Diamond\langle\tau\rangle_{\overline{z}}) \vee (\Diamond\Box\neg \mathit{en}(\langle\tau\rangle_{\overline{z}})) \end{array}$$

The weak fairness condition $WF_{\overline{z}}(\tau)$ says that from any point in an execution, the action τ must eventually be performed if it remains enabled until it is performed. The strong fairness condition $SF_{\overline{z}}(\tau)$ says that from any point in an execution, the action τ must be eventually executed infinitely often if it is infinitely often enabled.

The safety specifications $\Pi(P)$ and $\Phi(P)$ are usually strengthened by conjoining them with one or more fairness properties:

$$\Pi(P) \wedge \mathcal{L} \quad \text{and} \quad \exists \overline{x} : (\Pi(P) \wedge \mathcal{L})$$

5.3 Running example

The processor of a simple system issues *read* and *write* operations to be executed by the memory. The processor-memory interface has two registers, represented by the following state variables:

op: Set by the processor to the chosen operation, and reset by the memory after execution; its value space is $\{rdy, r, w\}$, for $ready$, $read$ and $write$, respectively.

val: Set by the processor to the value v to be written by a *write*, and by the memory to return the result of a *read*; its value space is the set of integers, **Z**.

Let the interface be a program P_1 with an (*internal*) variable d when denotes the data in the memory.

$$\begin{array}{rcll}
\overline{v}_1 & \triangleq & \{op, val, d\} & \\
\Theta_1 & \triangleq & (op \in \{rdy, r, w\}) \wedge d \in \mathbf{Z} & \text{initial condition} \\
R_1^p & \triangleq & (op = rdy) \wedge (op' = r) & \text{processor issues } read \\
W_1^p & \triangleq & (op = rdy) \wedge (op' = w) \wedge (val' \in \mathbf{Z}) & \text{processor issues } write \\
R_1^m & \triangleq & (op = r) \wedge (op' = rdy) \wedge (val' = d) & \text{memory executes } read \\
W_1^m & \triangleq & (op = w) \wedge (op' = rdy) \wedge (d' = val) & \text{memory executes } write \\
 & & & \\
A_1 & = & \{R_1^p, W_1^p, R_1^m, W_1^m\} & \text{actions of the program} \\
P_1 & = & (\overline{v}_1, \Theta_1, A_1) & \text{the program} \\
\mathcal{N}_{P_1} & = & R_1^p \vee W_1^p \vee R_1^m \vee W_1^m & \text{state transition relation} \\
\Pi(P_1) & = & \Theta_1 \wedge \Box[\mathcal{N}_{P_1}]_{\overline{v}_1} & \text{exact specification} \\
\Phi(P_1) & = & \exists d.\Theta_1 \wedge \Box[\mathcal{N}_{P_1}]_{\overline{v}_1} & \text{hiding the internal variable } d
\end{array}$$

The two actions R_1^p and W_1^p of the processor can be combined into a single nondeterministic action:

$$RW_1^p \triangleq (op = rdy) \wedge ((op' = r) \vee (op' = w) \wedge (val' \in \mathbf{Z}))$$

♣

5.4 Verification and Refinement

In TLA, verification of a program property specified by a formula φ which does not contain free internal variables is by proving the validity of the implication $\Phi(P) \Rightarrow \varphi$. A relatively complete proof system is given in [43], with additional rules for using the logic for reasoning about programs. Every valid TLA formula is provable from the axioms and proof rules of the TLA proof system if all the valid action formulas are provable. As the temporal operators $\Box$ and $\Diamond$ and the computational model are the same as those in [65], the rules and methods provided there for verification can be used.

Definition 5 *The relation $P_l \sqsubseteq P_h$ between two programs $P_l = (\overline{v}_l, \overline{x}, \Theta_l, A_l)$ and $P_h = (\overline{v}_h, \overline{y}, \Theta_h, A_h)$ characterizes* **refinement**, *i.e. that program P_l correctly implements P_h. Let*

$$\Phi(P_l) = \exists \overline{x}.\Theta_l \wedge \Box[\mathcal{N}_{P_l}]_{\overline{v}_l} \quad \textit{and} \quad \Phi(P_h) = \exists \overline{y}.\Theta_h \wedge \Box[\mathcal{N}_{P_h}]_{\overline{v}_h}$$

be canonical specifications of P_l and P_h respectively, where

$$\overline{x} = \{x_1, \ldots, x_n\} \qquad \overline{y} = \{y_1, \ldots, y_m\}$$

Then the refinement relation *is formalised as:*

$$P_l \sqsubseteq P_h \quad \textit{iff} \quad \Phi(P_l) \Rightarrow \Phi(P_h)$$

To prove the implication, we must define state functions $\tilde{y}_1, \ldots, \tilde{y}_m$ in terms of the variables $\overline{v}_l$ and prove the implication $\Pi(P_l) \Rightarrow \widetilde{\Pi(P_h)}$, where $\widetilde{\Pi(P_h)}$ is obtained from $\Pi(P_h)$ by substituting $\tilde{y}_i$ for all the free occurrences of y_i in $\Pi(P_h)$, for $i = 1, \ldots, m$. The collection of state functions $\tilde{y}_1, \ldots, \tilde{y}_m$ is called a *refinement mapping*. The substitutions can be applied also to a sub-formula of $\Pi(P_h)$. $\tilde{y}_i$ is the 'concrete' state function with which P_l implements the 'abstract' variable y_i of P_h. The proof of the implication can be carried out in two steps:

1. *initiality-preservation:* $\Theta_l \Rightarrow \widetilde{\Theta}_h$;
2. *step-simulation:* $\mathcal{N}_{P_l} \Rightarrow [\widetilde{\mathcal{N}}_{P_h}]_{\widetilde{\overline{v}}_l}$.

As $\mathcal{N}_{P_l}$ is the disjunction of the actions of P_l, step-simulation can be proved by showing $\tau \Rightarrow [\widetilde{\mathcal{N}}_{P_h}]_{\widetilde{\overline{v}}_l}$ for each $\tau \in A_l$; each step of the state transition by P_l corresponds to either a diligent step or a stuttering step by P_h.

5.4.1 Completeness remarks

The validity of the implication $\Phi(P_l) \Rightarrow \Phi(P_h)$ does not imply the existence of a refinement mapping, but in general, refinement mappings, can be found by adding dummy (or auxiliary) variables to specifications [2]. Once a refinement mapping is found, the verification of the refinement is straightforward and can be aided by mechanical means (e.g. [25]). However, finding a refinement mapping may be difficult if it is not known how P_l is obtained from P_h. On the other hand, knowing how an abstract state variable in P_h is implemented by the variables in P_l, it is possible to define the mapping between them. Refinement supports step-wise development in which a small number of abstract state variables are refined in each step.

5.5 Linking theories of programming to TLA

The use of atomic actions allows us to use most of the theories of sequential programming smoothly in this framework. In concurrent programming, an atomic action is often implemented as a *guarded command* which can be a big piece of program text [24] (also see Morgan's chapter on 'Probability in the context of wp' in this volume). A guarded command is of the form $g \longrightarrow C$, where C can be any programming statement such as

$C ::=$	$x := e$	
	$\mid\ C;\ C$	sequential composition
	$\mid\ C \sqcap C$	nondeterministic choice
	$\mid\ C \lhd b \lhd \rhd C$	conditional choice
	$\mid\ b * C$	iteration

For a given a command C, we can calculate a design $[\![C]\!]$ following the calculus of UTP [33]:

$$Pre_C \vdash Post_C$$

The corresponding TLA action of $g \longrightarrow C$ is then

$$g \wedge (Pre_C \Rightarrow Post_C)$$

We can use $(Pre_C \Rightarrow Post_C) \lhd g \rhd Skip$, as we allow stuttering.

Also, reasoning about TLA specifications, such as verifying an invariant property $\Box Q$, can be done by reasoning within UTP, Hoare Logic, or Dijkstra's Calculus of Weakest Preconditions. Refinement of a TLA specification can be carried out by refinement methods in UTP [33] or using the refinement calculi of Morgan [70] and Back [9].

Note that an atomic action does not have to be implemented by a sequential command. It can be implemented as a piece of concurrent program, say, written in Back's action systems [9].

6 Fault-tolerance

There are several different ways in which a program can be developed using formal rules which guarantee that it will *satisfy* a specification when executed on an fault-free system, e.g. [41, 9, 43]. However, when a component of a computer system fails, it will usually produce some undesirable effects and it can be said to no longer behave according to its specification. Such a breakdown of a component is called a fault and its consequence is called a failure. A fault may occur sporadically, or it may be stable and cause the component to fail permanently. Even when it occurs instantaneously, a fault such as a memory fault may have consequences that manifest themselves after a considerable time.

6.1 Introduction

Fault-tolerance is the ability of a system to function correctly despite the occurrence of faults. Faults caused by errors (or bugs) in software are systematic and can be reproduced in the right conditions. Formal methods can be used to address the problem of errors in software and, while their use does not guarantee the absence of software errors, they do provide the means of making a rigorous, additional check. Hardware errors may also be systematic but in addition they can have random causes. The fact that a hardware component functions correctly at some time is no guarantee of flawless future behaviour. Note that hardware faults often affect the correct behaviour of software.

Of course, it is not possible to tolerate every fault. A failure hypothesis stipulates how faults affect the behaviour of a system. An example of a failure hypothesis is the assumption that a communication medium might corrupt messages. With triple modular redundancy, a single component is replaced by three replicas and a voter that determines the outcome, and the failure hypothesis is that at any time at most one replica is affected by faults. A failure hypothesis divides abnormal behaviour, i.e. behaviour that does not conform to the specification, into exceptional and catastrophic behaviours. Exceptional behaviour conforms to the failure hypothesis and must be tolerated, but no attempt need be made to handle catastrophic behaviour (and, indeed, no attempt may be possible). For example, if the communication medium mentioned earlier repeatedly sends the same message, then this may be catastrophic for a given fault-tolerance scheme. It is important to note that normal behaviour does not mean perfect behaviour: after a time-out occurs, the retransmission of a message by a sender is normal but it may result in two copies of the same message reaching its destination. Exceptional and normal behaviours together form the acceptable behaviour that the system must tolerate.

Fault-tolerant programs are required for applications where it is essential that *faults* do not cause a program to have unpredictable execution behaviour. We assume that the failures do not arise from design faults in the program, since methods such as those mentioned above can be used to construct error-free programs. So, the only faults we shall consider are those caused by hardware and system malfunctions or the environment of the component that is under development. Many such failures can be *masked* from the program using automatic error detection and correction methods, but there is a limit to the extent to which this can be achieved at reasonable cost in terms of the resources and the time needed for correction.

When the nature or frequency of the errors makes automatic detection *and* correction infeasible, it may still be possible that error *detection* can be performed. It is desirable that fault-tolerant programs are able to perform predictably under these conditions: for example when using memory with single bit error correction and double bit error detection which operates even when the error correction is not effective. In fact, the provision of good program level fault-tolerance can make it possible to reduce the amount of expensive system error correction needed, as program level error recovery can often be focussed more precisely on the damage caused by an error than a general-purpose error correction mechanism.

The task is then to develop programs which perform predictably in the presence of *detected* system errors, and this requires the representation of such errors in the execution of a program. Earlier attempts to use formal proof methods for verifying the properties of fault-tolerant programs were based on an *informal* description of the effects of faults, and this limits their applicability. Here we shall instead model a fault as an *action* which performs state transformations in the same way as other program actions, making it possible to extend a semantic model to include fault actions and to use a single consistent method of reasoning for both program and fault actions.

Let P be a program satisfying the specification Sp. Let the effect of each physical fault in the system on which P is executed be described as a *fault action* which transforms a *good* program state into an *error* state which violates Sp. Physical faults are then modelled as the actions of a *fault program* F which interferes with the execution of P. A *failure* at any point during the execution of P takes it into an error state (F is assumed not to change an error state into a good state.).

In general a high level specification of a program is not sufficient to specify its behaviour in the presence of system faults or to transform it into a fault-tolerant program. It is also necessary to describe the hardware organisation of the system on which the program is to be executed, on its use of the resources of the system and the nature of the possible faults in the system, e.g. which processors and channels may fail; all of these factors can affect the execution of the program. Very little can be said about the effects of a system fault on a program until it has been refined to the level where these effects can be observed. There is need to represent faults and their effects at various levels of abstraction and here we shall use specifications to develop both the program and the fault environment in which it executes.

6.2 Formal specification, verification and refinement of fault-tolerant programs

A *physical fault* occurring during the execution of a program $P = (\overline{v}, \overline{x}, \Theta, A)$ can cause a transition from a valid state of P into an *error* state. This may lead to a *failure* state which violates the specification of P. A physical fault can be modelled as an atomic *fault-action*.

For example, a malicious fault may set the variables of P to arbitrary values, a crash in a processor may cause variables to become unavailable, and a fault may cause the loss of a message from a channel. Physical faults can thus be described by a set, F, of atomic actions which interfere with the execution of P by possibly changing the values of variables in $\overline{v}$. The *fault-environment* F can be specified by the action formula $\mathcal{N}_F$ which is the disjunction of the action formulas of all $\tau \in F$.

Executing $P = (\overline{v}, \overline{x}, \Theta, A)$ on a system with a fault-environment F is equivalent to interleaving the execution of the actions of P and F. Therefore, interference by F on the execution of P can be defined as a transformation $\mathcal{F}$:

$$\mathcal{F}(P, F) \triangleq (\overline{v}, \overline{x}, \Theta, A \cup F)$$

The exact and canonical specifications of the computations of P when executed on a system with faults F are given by:

$$\Pi(\mathcal{F}(P,F)) = \Theta \wedge \Box[\mathcal{N}_P \vee \mathcal{N}_F]_{\overline{v}} \qquad \text{and}$$
$$\Phi(\mathcal{F}(P,F)) = \exists \overline{x}.\Theta \wedge \Box[\mathcal{N}_P \vee \mathcal{N}_F]_{\overline{v}}$$

Definition 6 *The fault-prone properties of P under F can be derived from the properties of $\mathcal{F}(P, F)$, the F-**affected version** of P. A computation of $\mathcal{F}(P, F)$ is an F-**affected computation** of P.*

6.3 Running example continued

For the processor-memory interface, assume that the memory is faulty and that its value may be corrupted. Such a fault can be represented by the atomic operation

$$\textit{fault} \triangleq d' \neq d$$

Let the fault-environment F_1 contain the single action *fault*. The F_1-affected version of P_1 is then:

$$\mathcal{F}(P_1, F_1) = (\Theta_1, \{R_1^p, W_1^p, R_1^m, W_1^m, \textit{fault}\})$$

Thus, $\mathcal{N}_{F_1} = \textit{fault}$ and:

$$\begin{array}{lcl}
\mathcal{N}_{\mathcal{F}(P_1,F_1)} & = & \mathcal{N}_{P_1} \vee \textit{fault} \\
\Pi(\mathcal{F}(P_1,F_1)) & = & \Theta_1 \wedge \Box[\mathcal{N}_{\mathcal{F}(P_1,F_1)}]_{\overline{v}_1} \\
\Phi(\mathcal{F}(P_1,F_1)) & = & \exists d.\Pi(\mathcal{F}(P_1,F_1))
\end{array}$$

♣

For a program P to tolerate a set F of faults, *correcting actions* must be carried out to prevent an *error state* entered by a fault transition from leading to a *failure state* whose occurrence will violate the program requirement specification. In the example, the F_1-affected version $\mathcal{F}(P_1, F_1)$ of P_1 is not a refinement of P_1 and this implies that P_1 does not tolerate the fault F_1.

Definition 7 *For a given set F of faults, a program P is called a F-*tolerant implementation *of a property (or requirement) φ, if $\mathcal{F}(P, F)$ is an implementation of φ:*

$$\Phi(\mathcal{F}(P,F)) \Rightarrow \varphi$$

This means that the behaviours of P comply with the specification φ despite the presence of faults F. When such a property φ is a canonical specification of a program P_h,

$$\Phi(P_h) = \exists\overline{y}.\Theta_h \wedge \Box[\mathcal{N}_{P_h}]_{\overline{z}}$$

a program P_l is a *F-tolerant refinement* of P_h, denoted $P_l \sqsubseteq_F P_h$, if P_l is a F-tolerant implementation of $\Phi(P_h)$.

In general, a fault-tolerant program can be obtained from a fault-intolerant program P by [50, 53]

1. Adding *checkpointing operations*: $\mathcal{C}(P) = P_C$,
2. Adding *recovery operations*: $\mathcal{R}(\mathcal{C}(P)) = P_{FT}$.

C and $\mathcal{R}$ are required so that $\mathcal{F}(P_{FT}, F) \sqsubseteq P$ or $P_{FT} \sqsubseteq_F P$. There are other ways to construct a P_{FT} such that $P_{FT} \sqsubseteq_F P$.

In [50, 55], checkpointing actions and recovery actions are abstractly defined and can be refined to implement different kinds of fault-tolerant mechanisms.

The F-tolerant refinement relation $\sqsubseteq_F$ is stronger than the ordinary refinement relation: i.e. if P_l is a F-tolerant refinement of P_h, then P_l is a refinement of P_h but in general the converse is not true.

Further, F-tolerant refinement is generally not reflexive but it is transitive: if $P_{ll} \sqsubseteq_F P_l$ and $P_l \sqsubseteq_F P_h$, then $P_{ll} \sqsubseteq_F P_h$. Fault-tolerant refinements are *fault-monotonic*: if $\mathcal{N}_F \Rightarrow \mathcal{N}_{F_1}$ and $P_l \sqsubseteq_{F_1} P_h$, then $P_l \sqsubseteq_F P_h$. This means that a program which tolerates a set of faults also tolerates any subset of these faults [2].

Realistic modelling usually requires, in addition to the fault-actions, a *behavioural* fault assumption $\mathcal{B}_F$ about the global properties of F, such as the maximum number of memories corrupted at a time, and the minimum time between faults. This suggests that the exact specification of the F-affected computations of P should in general be specified as $\Pi(\mathcal{F}(P, F)) \wedge \mathcal{B}_F$, and the F-tolerant refinement of P_h by P_l should be proved under the condition $\mathcal{B}_F$:

$$\Pi(\mathcal{F}(P_l, F)) \wedge \mathcal{B}_F \Rightarrow \widetilde{\Pi(P_h)}$$

which is equivalent to $\mathcal{B}_F \Rightarrow (\Pi(\mathcal{F}(P_l, F)) \Rightarrow \widetilde{\Pi(P_h)})$. This indicates that the proof of F-tolerant refinement of P_h by P_l under $\mathcal{B}_F$ can be established by proving initiality-preservation and step-simulation under the assumption $\mathcal{B}_F$. A behavioural fault assumption prevents certain fault transitions from taking place from some states and is thus in general a safety property of the form $\Box\varphi$. Use the equivalence of $\Box\varphi_1 \wedge \Box\varphi_2$ and $\Box(\varphi_1 \wedge \varphi_2)$, the formula $\Box[\mathcal{N}_{P_l} \vee \mathcal{N}_F]_{\overline{v}} \wedge \mathcal{B}_F$ can be transformed into an equivalent formula $\Box[\mathcal{N}_1]_{\overline{v}}$. In fact, as $\mathcal{B}_F$ should not constrain the actions of P_l, $\mathcal{N}_1$ is obtained from $\mathcal{N}_{P_l}$ and $\mathcal{N}_F$ by enhancing the enabling conditions of the fault actions of F according to $\mathcal{B}_F$. For $\Pi(\mathcal{F}(P_l, F)) \wedge \mathcal{B}_F$, there is F_1 such that $\Pi(\mathcal{F}(P_l, F)) \wedge \mathcal{B}_F$ equals $\Pi(\mathcal{F}(P_l, F_1))$. This implies that the behavioural assumption $\mathcal{B}_F$ can be encoded into the set of fault actions and the two standard steps for proving refinement can be directly applied to the transformed specification $\Pi(\mathcal{F}(P_l, F_1))$. These two methods for proving a fault-tolerant refinement will be demonstrated in the example at the end of this section.

The separation of fault actions and behavioural assumptions simplifies the specification of the F-affected computations of program P_l. Further, coding these assumptions into the fault action makes the proof easier.

6.4 Running example continued

Let the fault-free memory of the processor-memory interface P_1 be implemented using three memories, such that at any time at most one suffers from faults.

Let d_i, $i = 1, 2, 3$, be the data in the three memories and let memory i be subject to *fault*$_i$. The variables f_i with value space $\{0, 1\}$ indicate that d_i has been corrupted when $f_i = 1$. The fault actions can be

[2]This is easily achieved in a linear time model, as with TLA. For a discussion of fault-monotonicity with a branching time model, see [35].

specified as follows:

$$\begin{array}{llll}
\mathit{fault}_i & = & (d_i' \neq d_i) \wedge (f_i' = 1) & \text{corrupts } d_i \\
F_2 & \triangleq & \{\mathit{fault}_1, \mathit{fault}_2, \mathit{fault}_3\} & \\
\mathcal{N}_{F_2} & = & \mathit{fault}_1 \vee \mathit{fault}_2 \vee \mathit{fault}_3 & \\
\mathcal{B}_{F_2} & \triangleq & \Box(f_1 + f_2 + f_3 \leq 1) & \text{at most one corrupted memory at any time}
\end{array}$$

Define the following auxiliary function:

$$\mathit{vote}(x, y, z) \triangleq \begin{cases} x & \text{if } x = y \text{ or } x = z \\ y & \text{if } x \neq y \text{ and } x \neq z \end{cases}$$

A program P_2 which tolerates the faults F_2 by using the $vote$ function to mask the corrupted copy of the memory, and its F_2-affected version are specified as follows:

$$\begin{array}{rcll}
\overline{v}_2 & \triangleq & \{op, val, d_1, d_2, d_3, f_1, f_2, f_3\} & \\
\Theta_2 & \triangleq & (op \in \{rdy, r, w\}) \wedge & \text{initially} \\
 & & (d_1 = d_2 = d_3) \wedge (\wedge_{i=1}^{3}(d_i \in \mathbf{Z})) & \text{all contain the same value} \\
R_2^p & \triangleq & (op = rdy) \wedge (op' = r) & \\
W_2^p & \triangleq & (op = rdy) \wedge (op' = w) \wedge (val' \in \mathbf{Z}) & \\
R_2^m & \triangleq & (op = r) \wedge (op' = rdy) \wedge & \\
 & & val' = \mathit{vote}(d_1, d_2, d_3) & \text{return the voted value} \\
W_2^m & \triangleq & (op = w) \wedge (op' = rdy) \wedge & \\
 & & \wedge_{i=1}^{3}(d_i' = val) \wedge & \text{write simultaneously} \\
 & & \wedge_{i=1}^{3}(f_i' = 0) & \text{overwrite corrupted copy} \\
 & & & \\
A_2 & = & \{R_2^p, W_2^p, R_2^m, W_2^m\} & \text{all actions} \\
P_2 & = & (\overline{v}_2, \Theta_2, A_2) & \text{program} \\
\mathcal{N}_{P_2} & = & R_2^p \vee W_2^p \vee R_2^m \vee W_2^m & \text{next-state relation} \\
\Pi(P_2) & = & \Theta_2 \wedge \Box[\mathcal{N}_{P_2}]_{\overline{v}_2} & \text{exact specification} \\
\Phi(P_2) & = & \exists(d_1, d_2, d_3, f_1, f_2, f_3).\Pi(P_2) & \text{canonical specification} \\
 & & & \\
\mathcal{F}(P_2, F_2) & = & (\overline{v}_2, \Theta_2, A_2 \cup F_2) & \text{fault-affected program} \\
\mathcal{N}_{\mathcal{F}(P_2, F_2)} & = & \mathcal{N}_{P_2} \vee \mathcal{N}_{F_2} & \\
\Pi(\mathcal{F}(P_2, F_2)) & = & \Theta_2 \wedge \Box[\mathcal{N}_{P_2} \vee \mathcal{N}_{F_2}]_{\overline{v}_2} & \\
\Phi(\mathcal{F}(P_2, F_2)) & = & \exists(d_1, d_2, d_3, f_1, f_2, f_3).\Pi(\mathcal{F}(P_2, F_2)) &
\end{array}$$

To prove the refinement relation $P_2 \sqsubseteq_{F_2} P_1$ under the assumption $\mathcal{B}_{F_2}$, define the mapping from the states of $\overline{v}_2$ to those of d: $\widetilde{d} = \mathit{vote}(d_1, d_2, d_3)$. Then, according to the definition of fault-tolerant refinement, we need to prove $\Pi(\mathcal{F}(P_2, F_2)) \wedge \mathcal{B}_{F_2} \Rightarrow \widetilde{\Pi(P_1)}$, where $\widetilde{\Pi(P_1)} = \Pi(P_1)[\tilde{d}/d]$, obtained by substituting $\tilde{d}$ for all occurrences of d in $\Pi(P_1)$.

Proof: [of the F_2-tolerant Refinement] The initiality-preservation $\Theta_2 \Rightarrow \widetilde{\Theta}_1$ holds trivially as $\tilde{d} = \textit{vote}(d_1, d_2, d_3)$, by definition. For step-simulation, we have:

Case 1: R_2^p and W_2^p, and R_2^m equal $\widetilde{R}_1^p$, $\widetilde{W}_1^p$ and $\widetilde{R_2^m}$, respectively;

Case 2: $W_2^m \Rightarrow \widetilde{W}_1^m$, as the right hand side is

$$(op = w) \wedge (op' = rdy) \wedge (\textit{vote}(d_1', d_2, d_3') = val')$$

Case 3: No $\textit{fault}_i$-step, for $i = 1, 2, 3$, changes the values val and op, and it is sufficient to show that no $\textit{fault}_i$-step changes $\tilde{d}$. We prove this for $i = 1$; the proofs for $i = 2, 3$ are similar. By the assumption $\mathcal{B}_{F_2}$ and the TLA rule for proving an invariance property, it is follows that $\mathcal{F}(P_2, F_2)$ has the following invariance property

$$\Box(f_i = 1 \Rightarrow (d_2 = d_3))$$

Thus, $\textit{fault}_1 \Rightarrow (d_2' = d_2) \wedge (d_3' = d_3) \wedge (d_2' = d_3')$ and this implies $\textit{fault}_1 \Rightarrow \textit{unchanged}(\tilde{d})$.

♡

We can also prove the fault-tolerant refinement as follows.

- First transform $\Theta_2 \wedge \Box[\mathcal{N}_{P_2} \vee \mathcal{N}_{F_2}]_{\overline{v}_2} \wedge \mathcal{B}_{F_2}$ into

  $$\Pi(\mathcal{F}(P_2, F_{21})) \triangleq \Theta_2 \wedge \Box[\mathcal{N}_{P_2} \vee \mathcal{N}_{F_{21}}]_{\overline{v}_2}$$

  where $F_{21} = \{\textit{fault}_{2i} : i = 1, 2, 3\}$ and

  $$\textit{fault}_{2i} \triangleq (f_{i\oplus 1} = 0 \wedge f_{i\oplus 2} = 0) \wedge (d_i' \neq d_i) \wedge (f_i' = 1)$$

  where $\oplus$ is $+$ modulo 3.

- Then prove $\Pi(\mathcal{F}(P_2, F_{21})) \Rightarrow \widetilde{\Pi(P_1)}$ by establishing initiality-preservation and step-simulation.

♣

7 Modelling real-time programs

The most common timing constraints over a program require its actions to be executed neither *too early* nor *too late*; for example, to use time for the synchronization between a processor and a memory to ensure that a message written is not overwritten before being read, the memory must not execute the $read$ operation too slowly and the processor must not issue the $write$ operation too soon. Let time be represented by the non-negative real numbers[3] $\mathbf{R}^+$. Timing constraints over the execution of an action in a program P can be specified by assigning to each action τ a volatile lower time bound $L(\tau)$ from $\mathbf{R}^+$ and a volatile upper time bound $U(\tau)$ which is either a value from $\mathbf{R}^+$, or the special value ∞ which denotes the absence of an upper bound. Any real number in $\mathbf{R}^+$ is assumed to be less than ∞, and the lower bound is assumed not to exceed the upper bound for any action. Both the lower and upper time bounds at the program level are volatile, and thus the semantic interpretation of L and U is that an action τ can be performed only if it has been *continuously* enabled for at least $L(\tau)$ time units; τ must be performed if it has been *continuously* enabled for $U(\tau)$ time units.

Definition 8 *A* **real-time program** *can be represented as a triple $P^T = \langle P, L, U\rangle$, where P is an 'untimed' program, defined in the previous section, and L and U are functions of the atomic actions of P defining the lower bound $L(\tau)$ and upper bound $U(\tau)$ for any action τ of P.*

7.1 Specifying real time

As in the case of untimed programs, we shall need an exact specification $\Pi(P^T)$ of a real-time program P^T. We introduce a distinguished state variable *now* to represent time, and an action to advance time, under the following assumptions [3, 30]:

time starts at 0: initially $\mathit{now} = 0$.

time never decreases: $\Box[\mathit{now}' \in (\mathit{now}, \infty)]_{\mathit{now}}$.

time diverges: $\forall t \in \mathbf{R}^+.\Diamond(\mathit{now} > t)$.

Time divergence is also called the Non-Zeno property and ensures that only a finite number of actions can be performed in any finite interval of time. The three assumptions can be combined to specify real-time evolution:

$$RT \triangleq (\mathit{now} = 0) \wedge \Box[\mathit{now}' \in (\mathit{now}, \infty)]_{\mathit{now}} \wedge \quad \forall t \in \mathbf{R}^+.\Diamond(\mathit{now} > t)$$

To preserve the atomicity of the actions in the program, we model the execution of the program so that program state and time do not change simultaneously and that a program state can be changed only by program actions, i.e. $\tau \Rightarrow (\mathit{now}' = \mathit{now})$ for each action τ of P. Then the conjunction $\Pi(P) \wedge RT$ specifies the interleaving of program actions and time evolution. The program actions are further constrained by their *lower bound* and *upper bound* conditions, and this is done by introducing auxiliary state variables called *timers*.

[3] The methods and results apply to discrete time domains as well.

7.2 Specifying time bounds

An action τ in P^T cannot take place before it has been enabled for $L(\tau)$ time units and must take place before it has been enabled for more than $U(\tau)$ units. We need to introduce auxiliary state variables to record how long an action has been enabled.

Consider the hour-clock again. Assume it is now required that the clock display the *correct real time*. Following what has been described earlier, we have:

- The newly added observable variable, *now*, representing time.
- The change of the display is instantaneous, e.g.

$$\left\{\begin{array}{l} hr \mapsto 12, \\ \mathit{now} \mapsto \sqrt{2.47} \end{array}\right\}, \left\{\begin{array}{l} hr \mapsto 12, \\ \mathit{now} \mapsto \sqrt{2.5} \end{array}\right\}, \ldots,$$

- now changes between of a change of display.
- The requirement that the interval between two ticks is one hour plus or minus ρ seconds.
- The need of a timer t to record how much time has elapsed since the last tick.

$$\begin{array}{lcl} \mathit{tNxt}(\mathit{HCnxt}) & \triangleq & (t' = 0) \lhd \mathit{HCnxt} \rhd (t' = t + (now' - now)) \\ Timer(t, HCnxt) & \triangleq & (t = 0) \land \Box[\mathit{tNxt}]_{(t,hr,now)} \end{array}$$

- The timer t cannot exceed $360 + \rho$ before the next tick:

$$\mathit{Max}(t, 360 + \rho) \triangleq \Box(t \leq 360 + \rho)$$

- After a tick, the clock cannot tick again before t becomes $360 + \rho$:

$$\mathit{Min}(t, \mathit{HCnxt}, 360 + \rho) \triangleq \Box[\mathit{HCnxt} \Rightarrow (t \geq 360 - \rho)]_{hr}$$

- The time bound specification HC_B is then the conjunction:

$$\mathit{HC}_B \triangleq Timer(t, HCnxt) \land \mathit{Max}t, 360 + \rho) \land \mathit{Min}(t, \mathit{HCnxt}, 360 + \rho)$$

The exact specification of the real-time clock is:

$$\mathit{RTHC} \triangleq \mathit{HC} \land \mathit{RT} \land \mathit{HC}_B$$

Definition 9 *In general, given a program $P = (\overline{v}, \overline{x}, \Theta, A)$, let $\tau \in A$ and δ be a non-negative real. We can define a* **counting-up volatile timer***:*

$$\begin{array}{lll} \mathit{Timer}(t_\tau, \tau) \triangleq & (t = 0) \wedge \Box[(t' = 0) \lhd (< \tau >_{\overline{v}} \vee \neg en(\tau)') \rhd \\ & (t'_\tau = t_\tau + (now' - now))]_{(t_\tau, \overline{v}, now)} \end{array}$$

where $A_1 \lhd g \rhd A_2$ denotes the action $g \wedge A_1 \vee \neg A_2$.

Then we can specify the time bounds of a real-time version P^T of program P as follows:

$$\begin{array}{lll} \mathit{Max} \uparrow (\tau) & \triangleq & \Box(t_\tau \leq U(\tau)) \\ \mathit{Min} \uparrow (\tau) & \triangleq & \Box[\tau \Rightarrow t \geq L(\tau)]_{(\overline{v}, now)} \\ B_\tau \uparrow & \triangleq & \mathit{Timer}(t_\tau, \tau) \wedge \mathit{Min} \uparrow (\tau) \wedge \mathit{Max} \uparrow (\tau) \\ B_P \uparrow & \triangleq & \bigwedge\limits_{\tau \in A} B_\tau \end{array}$$

The exact specification of P^T can be given as $\Pi(P) \wedge \mathit{RT} \wedge B_P \uparrow$.

Using counting-up timers gives a simpler specification of a real-time program. However, our experience is that with them, the proof of a refinement is hard. We now define a *counter-down timer*.

Definition 10 *Given a program $P = (\overline{v}, \overline{x}, \Theta, A)$, let $\tau \in A$ and δ be a non-negative real. We define* **volatile δ-timer** *t which is a state variable not in $\overline{v}$. The behaviour of the timer t is such that when τ is enabled from a state in which it was disabled or τ is taken, t is assigned a* clock time *of $now + \delta$ units of time:*

$$\begin{array}{ll} \mathit{Volatile}(t, \tau, \delta, \overline{v}) \triangleq & (((en(\tau) \wedge t = \delta)) \vee (\neg en(\tau) \wedge t = \infty)) \wedge \\ & \Box[\quad (en(\tau)' \wedge (\tau \vee \neg en(\tau)) \wedge (t' = now + \delta) \\ & \quad \vee en(\tau) \wedge en(\tau)' \wedge \neg\tau \wedge (t' = t) \\ & \quad \vee \neg en(\tau)' \wedge (t' = \infty)) \wedge (\overline{v}, now)' \neq (\overline{v}, now)]_{(t, \overline{v})} \end{array}$$

Informally, each line in the definition is explained as: the volatile δ-timer t is initially set to δ (i.e. δ time units ahead of the initial value 0 of *now*) if τ is enabled, and to ∞ otherwise, and then repeated in every step:

1. the timer t is reset to δ time units ahead of *now* in the new state if:
   (a) τ becomes enabled in the new state from being disabled in the old state, or
   (b) τ is taken and it remains enabled in the new state;
2. the timer t stays unchanged if τ remains enabled but τ has not taken place in this step;

3. the timer t is reset to ∞ if τ is disabled in the new state.

Using such a volatile timer t, the property that a τ-step cannot take place until the time *now* reaches the clock time t can be defined as:

$$\mathit{MinTime}(t, \tau, \overline{v}) \triangleq \Box[\tau \Rightarrow (t \leq \mathit{now})]_{\overline{v}}$$

The conjunction of this formula and *Volatile*$(t, \tau, \delta, \overline{v})$ can be used to specify a lower bound condition; and *Volatile*$(t, \tau, \delta, \overline{v})$ can be used also for an upper bound when conjoined with the formula:

$$\mathit{MaxTime}(t) \triangleq \Box[\mathit{now}' \leq t]_{\mathit{now}}$$

For a given real-time program $P^T = \langle P, L, U\rangle$, let each action τ of P have a volatile $L(\tau)$-timer t_τ and volatile $U(\tau)$-timer T_τ. Then the conjunction:

$$\mathit{Volatile}(t_\tau, \tau, L(\tau), \overline{v}) \wedge \mathit{MinTime}(t_\tau, \tau, \overline{v})$$

which is $true$ when $L(\tau) = 0$, specifies the lower bound for action τ. A τ-step cannot take place within $L(\tau)$ time units of when τ becomes enabled, and the next τ step cannot occur within $L(\tau)$ time units of when τ is re-enabled. The lower bound condition of the program is the conjunction of the lower bound conditions for all its actions:

$$LB(P^T) \triangleq \bigwedge_{\tau \in A} (\mathit{Volatile}(t_\tau, \tau, L(\tau), \overline{v}) \wedge \mathit{MinTime}(t_\tau, \tau, \overline{v}))$$

Similarly, the upper bound condition of program P^T is specified by the formula:

$$UB(P^T) \triangleq \bigwedge_{\tau \in A} (\mathit{Volatile}(T_\tau, \tau, U(\tau), \overline{v}) \wedge \mathit{MaxTime}(T_\tau))$$

where *Volatile*$(T_\tau, \tau, U(\tau), \overline{v}) \wedge$ *MaxTime*(T_τ) equals $true$ and thus can be eliminated from the conjunction if $U(\tau) = \infty$.

The time bound specification $B(P^T)$ for the whole program P^T is then the conjunction $LB(P^T) \wedge UB(P^T)$.

Definition 11 *The real-time executions of program P^T are exactly specified by the* **exact specification***:*

$$\Pi(P^T) \triangleq \Pi(P) \wedge RT \wedge B(P^T)$$

Hiding the internal variables $\overline{x}$ and the auxiliary timers, denoted by $timer(P^T)$, gives the **canonical specification** *of P^T:*

$$\Phi(P^T) \triangleq \exists \overline{x}, timer(P^T).\Pi(P^T)$$

7.3 Running example continued

In the untimed processor-memory interface P_1, let the processor and the memory be synchronised by timing rather than by guarding the processor actions. Assume that the processor periodically issues an operation every ρ units of time. To ensure that an operation is executed by the memory before the next operation is issued by the processor, ρ must be greater than the upper bound (or deadline) for the memory to execute the operation. The real-time program $P_1^T = \langle P_1, L_1, U_1 \rangle$ is described as follows:

$\overline{v}_1$	$\triangleq$	$\{op, val, d, c\}$	add an internal variable c
Θ_1	$\triangleq$	$(op \in \{r, w\}) \wedge (d \in \mathbf{Z}) \wedge \neg c$	the op has not been completed
RW_1^p	$\triangleq$	$(op' = r) \wedge \neg c' \vee$	issues a read operation, or
		$(op' = w) \wedge \neg c' \wedge (val' \in \mathbf{Z})$	a write operation
R_1^m	$\triangleq$	$(op = r) \wedge \neg c \wedge (val' = d) \wedge c'$	similar to the original P_1
W_1^m	$\triangleq$	$(op = w) \wedge \neg c \wedge (d' = val) \wedge c'$	similar to the original P_1
A_1	$\triangleq$	$\{RW_1^p, R_1^m, W_1^m\}$	actions of the program
$L_1(RW_1^p)$	$=$	$U_1(RW_1^p) = \rho$	RW_1^p's period
$L_1(R_1^m)$	$=$	$L_1(W_1^m) = 0$	memory actions' lower bound
$U_1(R_1^m)$	$=$	$U_1(W_1^m) = D_1 < \rho$	memory actions' upper bound
P_1^T	$=$	$\langle \overline{v}_1, \Theta_1, A_1, L_1, U_1 \rangle$	real-time program

♣

7.4 Verification and refinement

The timed and untimed properties of programs can be specified in the same way in TLA. For example, the bounded response property that once φ occurs in an execution, ψ must occur within δ time units can be described as:

$$\varphi \overset{\delta}{\leadsto} \psi \triangleq \forall t.\Box(\varphi \wedge \mathit{now} = t \Rightarrow \Diamond(\psi \wedge \mathit{now} \leq t + \delta))$$

To prove that the real-time program P^T *satisfies* (or *implements*) a timing property is to prove the implication of the property by the specification $\Phi(P)$ of the program. For example, the real-time processor-memory interface P_1^T satisfies the property:

$$\exists d.((op = r \wedge d = v) \overset{D_1}{\leadsto} (val = v))$$

which asserts that the value of the memory will be output within D_1 units of time after the processor issues a read operation. The implication:

$$\Phi(P_1^T) \Rightarrow \exists d.((op = r \wedge d = v) \overset{D_1}{\leadsto} (val = v))$$

can be proved by proving:

$$\Pi(P_1^T) \Rightarrow (op = r \wedge d = v) \overset{D_1}{\leadsto} (val = v)$$

Definition 12 *The* **refinement relation** $P_l^T \sqsubseteq P_h^T$ *between the real-time programs* P_l^T *and* P_h^T *is defined as the implication* $\Phi(P_l^T) \Rightarrow \Phi(P_h^T)$ *using a refinement mapping.*

To verify initiality-preservation and step-simulation, convert the exact specification:

$$\Pi(P^T) \triangleq \Theta_P \wedge [\mathcal{N}]_{\overline{v}} \wedge RT \wedge B(P^T)$$

into the form $\Theta \wedge [\mathcal{N}]_{\overline{z}}$, where $\overline{z}$ equals $\overline{v}$ plus *now* and the timers, and Θ is obtained from Θ_P by conjoining it with the initial conditions on *now* and the timers. $\mathcal{N}$ is an action formula.

8 Combining Fault-Tolerance and Timing Properties

Fault-tolerant systems often have real-time constraints. So it is important that the timing properties of a program are refined along with the functional and fault-tolerant properties defined in the program specification. This section extends the transformational approach for fault-tolerance by adding time bounds to actions. This will allow the fault-tolerant redundant actions to be specified with time constraints.

The functional properties of faults are modelled by a set F of atomic actions specified by the action formula $\mathcal{N}_F$. There are no time bounds on these actions (or, equivalently, the lower and upper bound of each fault action are respectively 0 and ∞). Given a real-time program $P^T = \langle P, L, U \rangle$, the F-affected version of P^T is defined as:

$$\mathcal{F}(P^T, F) \triangleq \langle \mathcal{F}(P, F), L, U \rangle$$

where the domain of L and U is extended to $A \cup F$ and each action in F is assigned time bounds of 0 and ∞.

To achieve fault-tolerance in a real-time system, there must be a timing assumption on the occurrence of faults, especially when deadlines are required to be met. Such an assumption is usually a constraint on the frequency of occurrence of faults, or the minimum time for which faults cannot occur. This period should be long enough for the recovery of the computation to take place and for progress to be made after recovery from a fault. For a formula ψ and a non-negative real number ε, let ψ hold continuously for ε units of time:

$$\Box_\varepsilon \psi \triangleq \forall t.(\mathit{now} = t \Rightarrow \Box(\mathit{now} \leq t + \varepsilon \Rightarrow \psi))$$

A fault is modelled as an atomic action and specified as an action formula. The timing assumption on faults F is a conjunction of assumptions, each of the form '*whenever fault*$_1$ *occurs, fault*$_2$ *cannot occur within* ε *units of time*'. If this assumption is denoted by $\mathcal{T}_F$, the exact and canonical specifications of the F-affected version $\mathcal{F}(P^T, F)$ are, respectively,

$$\begin{array}{rcl}
\Pi(\mathcal{F}(P^T, F)) & = & \Theta \wedge [(\mathcal{N}_P \vee \mathcal{N}_F)]_{\overline{v}} \wedge RT \wedge B(P^T) \wedge \mathcal{B}_F \wedge \mathcal{T}_F \\
 & = & \Pi(\mathcal{F}(P, F)) \wedge RT \wedge B(P^T) \wedge \mathcal{B}_F \wedge \mathcal{T}_F \\
\Phi(\mathcal{F}(P^T, F)) & = & \exists \overline{x}, timer(P^T).\Pi(\mathcal{F}(P^T, F))
\end{array}$$

Thus the F-affected version of a real-time program P^T is also a real-time program. This normal form allows the definition of *fault-tolerance for real-time systems* to be given in the same way as for untimed systems. A real-time program P^T is an F*-tolerant implementation* of a real-time property ψ if the implication $\Phi(\mathcal{F}(P^T, F)) \Rightarrow \psi$ holds. P^T is an F*-tolerant refinement* of a real-time program P_h^T if the implication $\Phi(\mathcal{F}(P^T, F)) \Rightarrow \Phi(P_h^T)$ holds.

8.1 Running example continued

In Section 6, we showed how the untimed fault-free processor-memory interface P_1 can be implemented by the untimed version of P_2, using three faulty memories whose values may be corrupted by the set F_2 of faults with assumption $\mathcal{B}_{F_2}$. We show now how the timed version P_1^T is F_2-tolerantly refined by a timed version of P_2.

Let the specification of the underlying untimed program P_2 be changed slightly by removing the guard condition of the processor actions:

$$\begin{array}{rcl}
\overline{v}_2 & = & \{op, val, c, d_1, d_2, d_3, f_1, f_2, f_3\} \\
\Theta_2 & = & (op \in \{r, w\}) \wedge \neg c \\
 & & (d_1 = d_2 = d_3) \wedge (\wedge_{i=1}^{3}(d_i \in \mathbf{Z})) \\
RW_2^p & = & (op' = r) \wedge \neg c' \vee (op' = w) \wedge \neg c' \wedge (val' \in \mathbf{Z}) \\
R_2^m & = & (op = r) \wedge \neg c \wedge c' \wedge val' = \mathit{vote}(d_1, d_2, d_3) \\
W_2^m & = & (op = w) \wedge \neg c \wedge c' \wedge \wedge_{i=1}^{3}(d_i' = val \wedge f_i' = 0) \\
\mathcal{N}_{P_2} & = & RW_2^p \vee R_2^m \vee W_2^m \\
\Pi(P_2) & = & \Theta_2 \wedge \Box[\mathcal{N}_{P_2}]_{\overline{v}_2} \\
\Phi(P_2) & = & \exists(d_1, d_2, d_3, c, f_1, f_2, f_3).\Pi(P_2)
\end{array}$$

Meeting the timing properties of P_1^T requires that the time bounds of the actions of the implementation P_2 guarantee (a) that the period for the processor to issue an operation is still ρ and (b) that the upper bound D_2 for the memory to execute an operation to completion is not greater than D_1:

$$\begin{array}{l} L_2(RW_2^p) = U_2(RW_2^p) = \rho \\ L_2(R_2^m) = L_2(W_2^p) = 0 \\ U_2(R_2^m) = U_2(W_2^m) = D_2 \leq D_1 \end{array}$$

To prove that $P_2^T \sqsubseteq_{F_2} P_1^T$ under the assumption $\mathcal{B}_{F_2}$, we have to consider only the case when $D_2 = D_1$ since simply lowering the upper bound (or raising the lower bound) of an action is obviously a refinement. Define a refinement mapping from the states over the variables of P_2^T's to the states over the internal variables of P_1^T, including the volatile timers as follows:

$$\begin{array}{lcllcl} \tilde{d} & \triangleq & \mathit{vote}(d_1, d_2, d_3) & \tilde{c} \triangleq c & & \\ \widetilde{t_{RW_1^p}} & \triangleq & t_{RW_2^p} & \widetilde{T_{RW_1^p}} & \triangleq & T_{RW_2^p} \\ \widetilde{T_{R_1^m}} & \triangleq & t_{R_2^m} & \widetilde{T_{W_1^m}} & \triangleq & T_{W_2^m} \end{array}$$

The implication $\Pi(P_2^T, F_2) \wedge \mathcal{B}_{F_2} \Rightarrow \widetilde{\Pi(P_1^T)}$ can be proved in the same way as for the untimed fault-tolerance in Section 6.

The assumption $\mathcal{B}_{F_2}$ can be relaxed to the timing assumption:

$$\mathcal{T}_{F_2} \triangleq \bigwedge_{i=1}^{3} \Box(f_i = 1 \Rightarrow \Box_{\rho+D_2}(f_{i\oplus 1} = 0 \wedge f_{i\oplus 2} = 0))$$

which asserts that only one of the most recently written memories may be corrupted before the read operation is completely executed. Then $P_2^T = \langle P_2, L_2, U_2 \rangle$ is also an F_2-tolerant refinement of P_1^T under the fault-assumption $\mathcal{T}_{F_2}$.

The specifications of P_1^T and P_2^T demonstrate a practical fact: to achieve fault-tolerance with timing constraints, a more powerful (or faster) machine is often needed. The execution of the multiple assignment W_2^m on such a machine should not be slower than the execution of the single assignment W_1^m on a machine for an non-fault-tolerant implementation of P_1^T; and the execution of the multiple read operation R_2^m with a voting function should not be slower than the execution of the single read operation R_1^m. Otherwise, with a machine of the same speed, the original time bounds must have enough slack to accommodate the redundant actions for fault-tolerance.

We can refine P_2^T further to P_3^T, where the actions of the three memories are executed by different processes, and the voting action is done by another process. The specification of the variables, the initial condition and the actions of P_3 are given below, for $i := 1, 2, 3$:

$$\begin{array}{rcl}
\overline{v}_3 & = & \{op, val, op_i, val_i, d_i, f_i, c_i, v_i \mid i = 1,2,3\} \\
\Theta_3 & = & (op \in \{r, w\}) \wedge \neg c_1 \wedge \neg c_2 \wedge \neg c_3 \wedge \\
 & & (op_1 = op_2 = op_3 = op) \wedge \neg v_1 \wedge \neg v_2 \wedge \neg v_3 \\
RW_3^p & = & \neg c_1' \wedge \neg c_2' \wedge \neg c_3' \wedge ((op, op_1, op_2, op_3)' = (r, r, r, r) \vee \\
 & & ((op, op_1, op_2, op_3)' = (w, w, w, w)) \wedge (val' \in \mathbf{Z})) \\
R_3^{m_i} & = & (op_i = r) \wedge \neg c_i \wedge (val_i' = d_i) \wedge c_i' \wedge v_i \\
W_3^{m_i} & = & (op_i = w) \wedge \neg c_i \wedge (d_i' = val) \wedge (f_i' = 0) \wedge c_i' \\
Vote & = & v_1 \wedge v_2 \wedge v_3 \wedge (val' = \mathsf{vote}(val_1, val_2, val_3) \wedge \\
 & & \neg v_1' \wedge \neg v_2' \wedge \neg v_3' \\
A_3 & = & \{RW_3^p, Vote, R_3^{m_i}, W_3^{m_i} \mid i = 1,2,3\} \\
P_3 & = & (\overline{v_3}, \Theta_3, A_3)
\end{array}$$

The newly introduced internal variables, op_i, contain the operations issued to the process i; c_i denote whether the operations issued to the process i are completed; v_i are used to synchronise the read actions and the vote action such that $vote$ is done only after all the $read$s are completed.

The timing properties of P_2^T require (a) that the time bounds of the actions in the implementation P_3^T guarantee that the period for the processor to issue an operation is still ρ, (b) that the upper bound D_{wi} for the ith memory to execute an issued write is not greater than D_2, and (c) that the sum of the upper bound D_{ri} of the ith memory to execute an issued read operation and the upper bound D_{vote} of the $Vote$ action is not greater than D_2: for $i = 1, 2, 3$.

$$\begin{array}{ll}
L_3(RW_3^p) = U_3(RW_3^p) = \rho & \\
L_3(R_3^{m_i}) = L_3(W_3^{m_i}) = L_3(Vote) = 0 & U_3(W_3^{m_i}) = D_{w_i} \\
U_3(R_2^{m_i}) = D_{r_i} & U_3(Vote) = D_{vote} \\
D_{w_i} \leq D_2 & D_{r_i} + D_{vote} \leq D_2
\end{array}$$

The refinement and fault-tolerance can be proved by showing the validity of the implication:

$$\Phi(\mathcal{F}(P_3^T, F_2)) \wedge \mathcal{B}_{F_2} \Rightarrow \Phi(\mathcal{F}(P_2^T, F_2)) \wedge \mathcal{B}_{F_2}$$

from the following refinement mapping when $D_{w_i} = D_2$ and $D_{r_i} + D_{vote} = D_2$:

$$\begin{array}{lcl}
\tilde{c} & \triangleq & c_1 \wedge c_2 \wedge c_3 \\
\tilde{d_i} & \triangleq & \begin{cases} d_i & \text{if } c_1 \wedge c_2 \wedge c_3 \vee \neg c_1 \wedge \neg c_2 \wedge \neg c_3 \\ val & \text{otherwise} \end{cases} \\
\widetilde{T_{R_2^m}} & \triangleq & \begin{cases} min\{T_{R_3^{m_i}} : i = 1,2,3\} + D_{vote} & \text{if } \vee_{i=1}^3 (op_i = r \wedge \neg v_i) \\ T_{Vote} & \text{if } v_1 \wedge v_2 \wedge v_3 \\ \infty & \text{otherwise} \end{cases} \\
\widetilde{T_{W_2^m}} & \triangleq & min\{T_{W_3^{m_i}} : i = 1,2,3\}
\end{array}$$

However, it is important to notice that it is easier to understand and prove the F_2-refinement of P_2^T by P_3^T if this refinement is done step-wise:

1. first refine P_2^T into a program P_{31}^T by replacing W_2^m in P_2 with the three write operations $W_3^{m_i}$ and setting $U(W^{m_i}) = D_2$, $i = 1, 2, 3$;
2. then refine P_{31}^T into another program P_{32}^T by replacing R_2^m in P_{31} (which is also in P_2) with the three read operations $R_3^{m_i}$ plus $Vote$ and setting $U(R^{m_i}) + U(Vote) = D_2$, $i := 1, 2, 3$;
3. finally, scale down the upper bounds of the new operations to get P_3^T.

♣

9 Feasible Real-Time Scheduling as Refinement

To model the parallel execution of a program P^T, we partition the actions A of P into n sets (*processes*) $p_1, \ldots, p_n$. A *shared state variable* is one which is used by actions in different processes, while a *private state variable* is used only by the actions in one process. Two actions of P can be executed in parallel *iff* they are not in the same process and do not share variables (shared variables are accessed under mutual exclusion). In such a concurrent system, processes communicate by executing actions which use shared variables. We assume that each process in a concurrent program is *sequential*, i.e. at most one atomic action in a process is enabled at a time, though an action of the process may be non-deterministic as it may carry out transitions from the same state to different states in different executions.

Let the real-time program P^T be implemented on a system by assigning its n processes to a set $\{1, \ldots, m\}$ of processors and executing them under a *scheduler*. Such an implementation is *correct* iff it meets both the functional requirements defined by the actions of P and the timing constraints defined by the time bound functions L and U of P^T. Rather than adding scheduling primitives to the programming (specification) language (e.g. as in [34]), the program and the scheduler will be modelled and specified in a single semantic model but their correctness will be proved separately. The application of a scheduler to a program on a given set of processors can be described as a transformation of the program, and the schedulability of the program can be determined by reasoning about the transformed or *scheduled* program.

Using transformations and separating the program from the scheduler helps to preserve the independence of the program from scheduling decisions. The programmer does not need to take account of the system and the scheduler until the program is ready to be implemented. This allows the feasibility of a program under different schedulers and the effect of a scheduler on different programs to be investigated. Also, the feasibility of the implementation of a program can be proved by considering a scheduling policy, rather than low-level implementation details.

We shall first describe the functional and timing aspects of a scheduler, and then determine how they affect the execution of the program.

9.1 Untimed scheduling

Assume that a scheduler allocates a process of P for execution by a processor using a *submit action*, and removes a process from a processor by a *retrieve action*. We shall say that a process is on a processor if the process has been allocated to that processor.

An atomic action of a process can be executed only when the process is on a processor and the action is enabled. Let the Boolean variable *run*$_i$, $1 \leq i \leq n$, be $true$ if process p_i is on a processor. The effect of scheduling is represented by a transformation $\mathcal{G}(P)$ in which each atomic action τ of P in the process p_i, $1 \leq i \leq n$, is transformed by strengthening its enabling condition by the Boolean variable *run*$_i$. Let $r(\tau)$ denote the transformed action of τ in $\mathcal{G}(P)$. Then:

$$r(\tau) \triangleq \mathit{run}_i \wedge \tau$$

Therefore, $\mathit{en}(r(\tau)) \Leftrightarrow \mathit{run}_i \wedge \mathit{en}(\tau)$, and a process p_i is *being executed* only when it is on a processor and one of its actions is enabled.

A scheduler can be functionally described as an untimed program S whose initial condition $\mathit{idle} \triangleq \forall i. \neg \mathit{run}_i$ guarantees that there is no process on any processor and whose submit and retrieve actions modify the variables *run*$_i$. We use a *generic* description so that the scheduler can be applied to any program P on a system with any number of processors. Program P and the set of processors will be left as parameters, to be replaced respectively by a concrete program and the definition of a specific system.

Given S, the scheduling of P by S on a set of processors can be described as a transformation $\mathcal{I}(P)$. The initial condition of the scheduled program $\mathcal{I}(P)$ is the conjunction of the initial conditions of S and P, i.e. $\mathit{idle} \wedge \Theta$. The actions of $\mathcal{I}(P)$ are formed by the union of the actions of S and $\mathcal{G}(P)$ and their execution is interleaved.

An execution of $\mathcal{I}(P)$ is a state sequence σ over the union of the state variables $\overline{z}$ of the scheduler and the variables $\overline{v}$ of the program P for which:

1. the initial state σ_0 satisfies the initial conditions Θ of P and *idle* of S,
2. for each step (σ_j, σ_{j+1}), one of the following conditions holds:
   (a) $\sigma_{j+1} = \sigma_j$, or
   (b) σ_{j+1} is produced from σ_j by an action in S, or
   (c) σ_{j+1} is produced by the execution of an action τ in a process p_i whose enabling condition and the predicate *run*$_i$ are both true in σ_j.

The set of executions of $\mathcal{I}(P)$ is then specified by:

$$\Pi(\mathcal{I}(P)) = \mathit{idle} \wedge \Theta \wedge \Box[\mathcal{N}_{\mathcal{G}(P)} \vee \mathcal{N}_S]_{(\overline{v},\overline{z})}$$

We assume that S does not change the state of P, i.e. $\mathcal{N}_S \Rightarrow (\overline{v}' = \overline{v})$. This gives us the compositional specification:

$$\Pi(\mathcal{I}(P)) = \Pi(S) \wedge \Pi(\mathcal{G}(P))$$

It can be seen that $r(\tau) \Rightarrow \tau$ holds for each action τ of P. So does $\Pi(\mathcal{G}(P)) \Rightarrow \Pi(P)$. Hence, $\Pi(\mathcal{I}(P))$ implies $\Pi(P)$. This shows that $\mathcal{I}(P)$ refines P and the transformation $\mathcal{I}$ (and thus the scheduler S) preserves the functional properties of P.

9.2 Timed scheduling

The timing properties of the executions of $\mathcal{I}(P)$ depend on the number of processors and their execution speed. Assume that the *hard execution time* needed for each atomic operation τ on a processor lies in a real interval $[l(\tau), u(\tau)]$: i.e. if the execution of τ on a processor starts at time t and finishes at time $t + d$, then the *total execution time* for τ in the interval $[t, t + d]$ lies in the interval $[l(\tau), u(\tau)]$. The functions l and u define the (persistent) time bounds of the actions in $\mathcal{G}(P)$. The real-time program $\mathcal{G}(P)^T \triangleq \langle \mathcal{G}(P), l, u \rangle$, where for each $r(\tau)$ of $\mathcal{G}(P)$, $l(r(\tau)) = l(\tau)$ and $u(r(\tau)) = u(\tau)$.

To guarantee that the implementation of P^T satisfies its real-time deadlines, the computational overhead of the *submit* and *retrieve* actions must be bounded. Let the scheduler S have time bounds $L_S(\tau)$ and $U_S(\tau)$ for each action τ of S and let the real-time scheduler be S^T.

Definition 13 *The* **real-time scheduled program** $\mathcal{I}(P^T) \triangleq \langle \mathcal{I}(P), L_{\mathcal{I}(P)}, U_{\mathcal{I}(P)} \rangle$*, where the functions* $L_{\mathcal{I}(P)}$ *and* $U_{\mathcal{I}(P)}$ *are respectively the union*[4] *of the functions* L_S *and* l*, and the union of* U_S *and* u*.*

This means that the execution speed of the processors and the timing properties of the scheduler determine the timing properties of the scheduled program.

As the actions of the scheduler are not interrupted, the time bounds L_S and U_S of actions of S are volatile. However, an execution of a process action may be pre-empted, e.g. under a priority-based pre-emptive scheduler. Thus, the time bounds l and u for the actions in $\mathcal{G}(P)$ should be persistent in general. Moreover, in a concurrent program, a pre-empted action may be disabled by the execution of the actions of other processes. When the pre-empted process is resumed, this pre-empted (and disabled) action will not be executed and another enabled action in this process will be selected for execution. For this reason, we need the notion of a persistent timer.

[4]For functions p from Set_1 to Set and q from Set_2 to Set, where Set_1 and Set_2 are disjoint, the *union* of p and q is the function from $Set_1 \cup Set_2$ to Set that equals p for elements in Set_1 and q for elements in Set_2.

Definition 14 *A* **persistent** δ**-timer** t *for an action* τ *in process* p_i *is defined as follows:*

$$\begin{array}{rlr} \mathit{Persistent}(t,\tau,\delta,\overline{v}) \triangleq & t=\delta \wedge & \\ & \Box[\quad (r(\tau)\wedge t'=\mathit{now}+\delta & \mathit{taken} \\ & \quad \vee\ \mathit{en}(r(\tau))\wedge\neg r(\tau)\wedge t'=t & \mathit{running} \\ & \quad \vee\ \neg\mathit{en}(\tau)'\wedge t'=\mathit{now}'+\delta & \mathit{disabled} \\ & \quad \vee\ \mathit{en}(\tau)\wedge\neg\mathit{run}_i\wedge t'=t+(\mathit{now}'-\mathit{now})) & \mathit{pre\text{-}empted} \\ & \wedge((\overline{v},\mathit{now})'\neq(\overline{v},\mathit{now}))]_{(t,\overline{v},\mathit{now})} & \end{array}$$

Informally,

1. the persistent δ-timer t is initially (i.e. when $\mathit{now}=0$) set to δ;
2. it stays δ time units ahead of *now* as long as τ is not enabled (i.e. $\neg\mathit{en}(\tau)$ holds);
3. it remains unchanged during any time when τ is both enabled and run (i.e. $\mathit{en}(r(\tau))$ holds) to record the execution time;
4. it is reset either just after a τ-step is taken or τ is disabled; and
5. it changes at the same rate as *now* when τ is enabled but not run (i.e. $\mathit{en}(\tau)\wedge\neg\mathit{run}_i$ holds), i.e. the time when a process is waiting for the processor or the execution of τ is pre-empted should not be counted as execution time.

Conditions (4) & (5) guarantee that timer t is persistent only when τ is pre-empted, and if τ is pre-empted the intermediate state at the point of pre-emption is not observable to other actions.

The conjunction of the defining formula of a persistent $u(\tau)$-timer T_τ for action τ and $\mathit{MaxTime}(u(\tau))$:

$$\mathit{Persistent}(T_\tau,\tau,u(\tau),\overline{v})\wedge\mathit{MaxTime}(T_\tau)$$

is the specification of the upper persistent time bound condition for action $r(\tau)$, and this asserts that the τ-step of state transition must take place if the accumulated time when τ has been both enabled and run reaches $u(\tau)$. Similarly, the lower persistent time bound condition for action τ is specified by:

$$\mathit{Persistent}(t_\tau,\tau,l(\tau),\overline{v})\wedge\mathit{MinTime}(t_\tau,r(\tau),\overline{v})$$

Notice that when there is no pre-emption in the execution of the program, i.e.:

$$\Box(\mathit{en}(r(\tau))\wedge\neg\mathit{run}_i'\Rightarrow(\neg\mathit{en}(\tau)\vee r(\tau)))$$

is ensured by the scheduler; the use of a persistent timer of τ in these two formulas is equivalent to the use of a volatile timer of $r(\tau)$:

1. *Persisten*$(t, \tau, \delta, \overline{v})$ initially sets t to δ, and keeps resetting t with *now* $+ \delta$ as long as $\neg$*en*$(r(\tau))$. This is the same as in *Volatile*$(t, r(\tau), \delta, \overline{v})$ which sets t to ∞ and keeps it unchanged until *en*$(r(\tau))$ becomes $true$ and sets it to *now*$' + \delta$.

2. Assume *en*$(\tau) \wedge \neg$*run*$_i$ has been $true$ since, say *now* $=$ *now*$_0$, and t was set by *Persisten*$(t, \tau, \delta, \overline{v})$ to *now*$_0 + \delta$. From *now*$_0$, *Persisten*$(t, \tau, \delta, \overline{v})$ increases t at the same rate by which *now* increases as there cannot be a pre-emption. This is the same as in *Volatile*$(t, r(\tau), \delta, \overline{v})$ where t was set to ∞ and kept unchanged unless *run*$_i$ becomes $true$ when t is set to *now*$' + \delta$.

Thus, persistent timers allow the treatment of both pre-emptive and non-pre-emptive scheduling.

The specification of the timing condition for $\mathcal{G}(P)^T$ is defined as

$$\begin{aligned} B(\mathcal{G}(P)^T) \quad \triangleq \quad & \bigwedge_{\tau \in A} (\textit{Persistent}(t_\tau, \tau, l(\tau), \overline{v}) \wedge \textit{MinTime}(t_\tau, r(\tau), \overline{v})) \wedge \\ & \bigwedge_{\tau \in A} (\textit{Persistent}(T_\tau, \tau, u(\tau), \overline{v}) \wedge \textit{MaxTime}(T_\tau)) \end{aligned}$$

The exact specification of the timed scheduled program $\mathcal{I}(P^T)$ is

$$\begin{aligned} \Pi(\mathcal{I}(P^T)) \quad & = \Pi(S) \wedge \Pi(\mathcal{G}(P)) \wedge RT \wedge B(S^T) \wedge B(\mathcal{G}(P)^T) \\ & = \Pi(S^T) \wedge \Pi(\mathcal{G}(P)^T) \end{aligned}$$

The correctness of the timed scheduled program $\mathcal{I}(P^T)$ is determined with respect to the specification of P^T, which does not refer to the variables $\overline{z}$ which are modified by the scheduler S. These variables (and those which are internal to S) are therefore hidden in the canonical specification

$$\Phi(\mathcal{I}(P^T)) \triangleq \exists \overline{z}.\Phi(S^T) \wedge \Phi(\mathcal{G}(P)^T)$$

We shall use this specification in the following section where we consider two ways of applying the transformational approach to real-time scheduling.

9.3 Reasoning about scheduled programs

Consider the implementation of a real-time program P^T using a real-time scheduler S^T which satisfies a property φ. Proof that this implementation satisfies a high-level timing property ψ, whose only free state variables are *now* and the external variables of S, can be used as the initial basis from which proofs of more detailed low level properties can later be established.

Because of the assumption that the program and the scheduler do not change the same variables, if S^T satisfies a property φ and $\varphi \wedge \Phi(\mathcal{G}(P)^T)$ implies ψ, then $\mathcal{I}(P^T)$ satisfies ψ. This is represented as the proof rule:

$$R1. \quad \frac{\begin{array}{ll} 1 & \Phi(S^T) \Rightarrow \varphi \\ 2 & \exists \overline{z}.\varphi \wedge \Phi(\mathcal{G}(P)^T) \Rightarrow \psi \end{array}}{\Phi(\mathcal{I}(P^T)) \Rightarrow \psi}$$

Treating the effect of scheduling as a transformation of a program specification allows an abstract specification of a scheduler's *policy* to be used to prove the timing properties of the implementation of a real-time program.

9.4 Feasibility: definition and verification

Definition 15 *The timed scheduled program $\mathcal{I}(P^T)$ is* **feasible** *if $\Phi(\mathcal{I}(P^T)) \Rightarrow \Phi(P^T)$, i.e. if there is a refinement mapping by which the following implication can be proved:*

$$\Pi(\mathcal{I}(P^T)) \Rightarrow \widetilde{\Pi(P^T)}$$

Notice that the correctness of a scheduler is defined with respect to its specification (or its scheduling policy) while feasibility relates the specification of the program P^T to be scheduled to the specification of the scheduled program and requires the time bounds of all actions of the former to be met by the later. Assuming that $\Phi(S^T) \Rightarrow \varphi$, the feasibility of $\mathcal{I}(P^T)$ can be proved from Rule R1 as the implication:

(1) $\quad \exists \overline{z}.\varphi \wedge \Phi(\mathcal{G}(P)^T) \Rightarrow \Phi(P^T)$

This formula can be manipulated in steps, using a refinement mapping.

Step 1 *Introduce auxiliary (dummy) timers into $\mathcal{G}(P)^T$ corresponding to the timers of P^T.*

This can be understood as allowing the scheduler to have a copy of the timers of $P^T = \langle P, L, U \rangle$. Define a set of auxiliary variables:

$$\textit{dummies} \triangleq \{h_\tau, H_\tau \mid \tau \in A\}$$

where h_τ and H_τ are respectively defined by the formulas *Volatile*$(h_\tau, \tau, L(\tau), \overline{v})$ and *Volatile*$(H_\tau, \tau, U(\tau), \overline{v})$. Let:

$$D(\textit{dummies}) \triangleq \bigwedge_{\tau \in A} \textit{Volatile}(h_\tau, \tau, L(\tau), \overline{v}) \wedge \textit{Volatile}(H_\tau, \tau, U(\tau), \overline{v})$$

Then (1) is equivalent to:

(2) $\exists \mathit{dummies}, \overline{z}.\varphi \wedge \Phi(\mathcal{G}(P)^T) \wedge D(\mathit{dummies}) \Rightarrow \Phi(P^T)$

Step 2 *Define the refinement mapping.*

Recall that the internal variables of P are assumed to be $\overline{x}$. A *refinement mapping* from the states over $\overline{x} \cup \overline{z} \cup timer(\mathcal{G}(P)^T) \cup \mathit{dummies}$ to the states over $\overline{x} \cup timer(P^T)$ is defined as follows:

$$\tilde{y} = \begin{cases} h_\tau & \text{if } y \text{ is a timer } t_\tau \in timer(P^T) \\ H_\tau & \text{if } y \text{ is a timer } T_\tau \in timer(P^T) \\ y & \text{if } y \in \overline{x} \end{cases}$$

Let *TimedSched* $\triangleq \varphi \wedge \Pi(\mathcal{G}(P)^T) \wedge D(\mathit{dummies})$. Then (2) can be proved by proving:

(3) $\mathit{TimedSched} \Rightarrow \widetilde{\Pi(P^T)}$

Step 3 *Discard identical substitutions.*

Recall that $\widetilde{\Pi(P^T)} = \widetilde{\Pi(P)} \wedge \widetilde{RT} \wedge \widetilde{B(P^T)}$. Obviously, $\widetilde{RT} = RT$ and $\widetilde{\Pi(P)} = \Pi(P)$. Also $\Pi(\mathcal{G}(P)^T)$ implies $\Pi(\mathcal{G}(P))$, which in turn implies $\Pi(P)$. Therefore, $\widetilde{RT}$ and $\widetilde{\Pi(P)}$ can be discarded from the right hand side of the implication in (3).

(4) $\mathit{TimedSched} \Rightarrow \widetilde{B(P^T)}$

Step 4 *Discard the actions on timers.*

$$\begin{aligned} \widetilde{B(P^T)} &= \bigwedge_{\tau \in A} \mathit{Volatile}(h_\tau, \tau, L(\tau), \overline{v}) \wedge \mathit{MinTime}(h_\tau, \tau, \overline{v}) \wedge \\ &\qquad \mathit{Volatile}(H_\tau, \tau, U(\tau), \overline{v}) \wedge \mathit{MaxTime}(H_\tau) \\ &= D(\mathit{dummies}) \wedge \bigwedge_{\tau \in A} (\mathit{MaxTime}(H_\tau) \wedge (\mathit{MinTime}(h_\tau, \tau, \overline{v})) \end{aligned}$$

Since $D(\mathit{dummies})$ appears on the left hand side of (4), what remains to be proved is that the following implication holds for each action τ of P.

(5) $\mathit{TimedSched} \Rightarrow \mathit{MaxTime}(H_\tau) \wedge \mathit{MinTime}(h_\tau, \tau, \overline{v})$

9.5 Proof rules for feasibility

Implication 5 suggests that the feasibility of an implementation of a real-time program P^T can be proved using the following rule:

$$R2.\quad \dfrac{\begin{array}{ll} 1 & \Phi(S^T) \Rightarrow \varphi \\ 2 & \textit{TimedSched} \Rightarrow \textit{MaxTime}(H_\tau) \text{ for } \tau \in A \\ 3 & \textit{TimedSched} \Rightarrow \textit{MinTime}(h_\tau, \tau, \overline{v}) \text{ for } \tau \in A \end{array}}{\Phi(\mathcal{I}(P^T)) \Rightarrow \Phi(P^T)}$$

Notice that both $\textit{MaxTime}(H_\tau)$ and $\textit{MinTime}(h_\tau, \tau, \overline{v})$ contain primed state variables. Therefore, rules for proving invariant properties cannot be used directly to establish the premises (2) and (3) in Rule R2. We provide two rules for introducing invariants.

To prove premise (2) in Rule R2, we have the following rule:

$$R3.\quad \dfrac{\begin{array}{ll} 1 & \Phi(S^T) \Rightarrow \varphi \\ 2 & \textit{TimedSched} \Rightarrow \Box(\textit{en}(\tau) \Rightarrow T_\tau \leq H_\tau) \text{ for } \tau \in A \end{array}}{\textit{TimedSched} \Rightarrow \textit{MaxTime}(H_\tau) \text{ for } \tau \in A}$$

By symmetry, for premise (3) in Rule R2:

$$R4.\quad \dfrac{\begin{array}{ll} 1 & \Phi(S^T) \Rightarrow \varphi \\ 2 & \textit{TimedSched} \Rightarrow \Box(\textit{run}_i \Rightarrow t_\tau \geq h_\tau) \text{ for } \tau \text{ in } p_i \end{array}}{\textit{TimedSched} \Rightarrow \textit{MinTime}(h_\tau, \tau, \overline{v}) \text{ for } \tau \in A}$$

TimedSched can be converted into a normal form as the conjunction of a safety property and a liveness property:

$$\exists \overline{x}. \Theta \wedge \Box[\mathcal{N}]_{\overline{y}} \wedge \mathcal{L}$$

where $\overline{x}$ and $\overline{y}$ are sets of variables, Θ is a state predicate, $\mathcal{N}$ is an action and $\mathcal{L}$ is the time divergence property, $\forall t. \Diamond(\textit{now} > t)$.

Let this formula be denoted by Ω. An invariant Q of Ω can be proved using the rule:

$$R5.\quad \dfrac{\begin{array}{lll} 1 & \Theta \Rightarrow Q & \text{Initially } Q \text{ holds} \\ 2 & Q \wedge \mathcal{N} \Rightarrow Q' & \text{Each step of the transition preserves } Q \end{array}}{\Omega \Rightarrow \Box Q}$$

9.6 Feasibility of fault-tolerant real-time programs

The occurrence of a fault-action does not depend on the scheduler and the F-affected scheduled program of P^T by a scheduler S is modelled as $\mathcal{F}(\mathcal{I}(P^T), F)$ whose exact specification is:

$$\Pi(\mathcal{F}(\mathcal{I}(P^T), F)) = \Pi(S) \wedge \Pi(\mathcal{F}(\mathcal{G}(P), F)) \wedge RT \wedge B(S^T) \wedge B(\mathcal{G}(P^T))$$

Let *TimedSched* (from the previous subsection) be redefined as:

$$\mathit{TimedSched} \triangleq \varphi \wedge \Pi(\mathcal{F}(\mathcal{G}(P^T), F)) \wedge D(\mathit{dummies})$$

Definition 16 *Taking the same set of dummy variables dummies and the refinement mapping from the previous subsection, the implementation $\mathcal{I}(P^T)$ is F-**tolerantly feasible** if the following implication holds:*

$$\mathit{TimedSched} \Rightarrow \Pi(\widetilde{\mathcal{F}(P^T}, F))$$

Then all the equations and rules in the previous section remain valid for fault-tolerant feasibility.

Assume that a real-time program $P^T = \langle P, L, U \rangle$ is a F-tolerant refinement of a program P_h^T for a given set F of fault-actions. Then any F-tolerant feasible implementation of P^T is a F-tolerant refinement of P_h^T.

This assumes that the execution of the scheduler is not faulty, and F-tolerance is provided by the program to be scheduled. It is also possible for a non-fault-tolerant or a fault-tolerant program to be executed under a specially designed scheduler so that the implementation of the faulty program is fault-tolerant [55]. For example, a scheduler can be designed to tolerate processor failures. Assume each process of P^T keeps taking checkpoints of its local states by a transformation $\mathcal{C}(P^T)$. We add recovery process(es) to $\mathcal{C}(P^T)$ by a transformation $\mathcal{R}(\mathcal{C}(P^T))$. Faults and their effects on processes are modelled as before. The implementation transformation $\mathcal{I}$ is applied to $\mathcal{F}(\mathcal{R}(\mathcal{C}(P^T)), F)$. When a processor fails, the scheduler must submit the recovery process to a non-failed processor. If the processors are *fail-stop*, no check-pointing or recovery may be needed. The scheduler only needs to re-schedule a process executing on a failed processor to a non-failed processor, where that is possible.

9.7 Scheduling open systems

In the model of programs given so far, we have assumed that a real-time program implements the specification of a *closed system*: values are supplied to the program through the initial values of variables or by executing a nondeterministic *input* operation.

In many cases, a program is linked to an external environment from which it receives data and to which it must send responses. The appearance of the inputs often follows a timing pattern, for example with *periodic* or *aperiodic* repetition.

Definition 17 *An* **open system** *is a pair $O = (E, P)$ consisting of a program P which interacts with an environment E. The set $\overline{v}_o$ of variables of O is the union of the sets $\overline{x}$ and $\overline{y}$ of* local variables *of P and E and the set $\overline{v}$ of* interface variables *through which P and E interact.*

Let program P consist of an initial predicate $\Theta_{\overline{x}}$ over its local variables $\overline{x}$ and a set of atomic actions on the program variables $\overline{v}_p = \overline{x} \cup \overline{v}$. Let the environment E consist of an initial predicate Θ over the environment variables $\overline{v}_e = \overline{y} \cup \overline{v}$ and a set of atomic actions on the variables $\overline{v}_e$.

Let ν be an action formula that defines the state transitions by which P changes the values of the interface variables. It is then required [3] that:

$$\mathcal{N}_P \Rightarrow \nu \vee (\overline{v}' = \overline{v}) \quad \text{and} \quad \mathcal{N}_E \Rightarrow \neg\nu \vee (\overline{v}' = \overline{v})$$

As before, we define:

$$\Pi(P) \triangleq \Theta_{\overline{x}} \wedge \Box[\neg\nu \wedge (\overline{x}' = \overline{x}) \vee \mathcal{N}_P]_{\overline{v}_p} \quad \text{and} \quad \Phi(P) \triangleq \exists \overline{x}.\Pi(P)$$

$$\Pi(E) \triangleq \Theta \wedge \Box[\nu \wedge (\overline{y}' = \overline{y}) \vee \mathcal{N}_E]_{\overline{v}_e} \quad \text{and} \quad \Phi(E) \triangleq \exists \overline{y}.\Pi(E)$$

The specification $\Phi(O)$ of an open system $O = (E, P)$ then defines the condition under which the system guarantees the property $\Phi(P)$ if the environment satisfies the assumption $\Phi(E)$.

$$\Phi(O) \triangleq \Phi(E) \Rightarrow \Phi(P)$$

The conjunction $\Phi(E) \wedge \Phi(P)$ describes the closed system consisting of P and its environment E and is:

$$\exists \overline{x}, \overline{y}.\Theta \wedge \Theta_{\overline{x}} \wedge \Box[\mathcal{N}_P \vee \mathcal{N}_E]_{\overline{v}_o}$$

Program P_l refines (or implements) a program P_h in environment E iff:

$$(\Phi(E) \Rightarrow \Phi(P_l)) \Rightarrow (\Phi(E) \Rightarrow \Phi(P_h))$$

and this reduces to:

$$\Phi(E) \wedge \Phi(P_l) \Rightarrow \Phi(P_h)$$

The program and its environment can be treated as the real-time programs $P^T = \langle P, L, U\rangle$ and $E^T = \langle E, L_e, U_e\rangle$ respectively. Since time is global, it need not be advanced by both of them. We choose to let the program advance time and define:

$$\Phi(E^T) \triangleq \exists \overline{y}, timer(E^T).\Pi(E) \wedge B(E^T)$$

The *real-time open system* $O^T = (E^T, P^T)$ is specified by[5]:

$$\Phi(O^T) \triangleq \Phi(E^T) \Rightarrow \Phi(P^T)$$

A real-time property φ of an open system $O^T = (E^T, P^T)$ states that program P^T guarantees the property φ under the environment assumption E^T. This requires proving the implication:

$$\Phi(O^T) \Rightarrow (\Phi(E^T) \Rightarrow \varphi)$$

or, equivalently,

$$\Phi(E^T) \wedge \Phi(P^T) \Rightarrow \varphi$$

In a real-time environment E^T, implementation of a real-time program P^T by a scheduler S^T on a set of processors can be described by transformation $\mathcal{I}(O^T) \triangleq (E^T, \mathcal{I}(P^T))$, in which $\mathcal{I}(P^T)$ is as defined in Section 4.2 for a closed system, and $\overline{z}$ denotes the variables that may be changed by the scheduler.

The feasibility of the implementation relies on proving the refinement relation: $\mathcal{I}(O^T) \sqsubseteq O^T$ i.e. the implication

$$\Phi(\mathcal{I}(O^T)) \Rightarrow \Phi(O^T)$$

or equivalently on proving:

$$(6) \qquad \Phi(E^T) \wedge \Phi(\mathcal{I}(P^T)) \Rightarrow \Phi(P^T)$$

It is easy to see that Rules R1–R5 apply also to open systems.

[5]The canonical form of an open real-time specification given here is simpler than that in [3] but is sufficient for our purposes as we shall not be considering the problem of composing open systems.

9.8 Running example continued

In the timed fault-tolerance processor-memory interface program P_3^T, let RW_3^p be an environment action with its lower and upper bounds (i.e. period) set to ρ. Partition the remaining actions into four processes:

$$\begin{array}{lll}
p_4 = \{Vote\} & p_i = \{R_3^{m_i}, W_3^{m_i}\} & \text{for } i = 1,2,3 \\
L_3(Vote) = 0 & L_3(R_3^{m_i}) = L_3(W_3^{m_i}) = 0 & \text{for } i = 1,2,3 \\
U_3(R_3^{m_i}) = D_{r_i} & U_3(W_3^{m_i}) = D_{w_i} \leq D_2 & \text{for } i = 1,2,3 \\
U_3(Vote) = D_{vote} & D_{r_i} + D_{vote} \leq D_2 & \text{for } i = 1,2,3
\end{array}$$

where D_2 is the deadline of the memory actions in the real-time interface program P_2^T implemented by P_3^T.

Let the memory processes be implemented on a single processor using a non-deterministic scheduler. Ignore the details of the scheduler program: e.g. assume that it randomly chooses an enabled process. If there is no overhead in the scheduling, the scheduler can be specified as a real-time program $S^T = \langle S, L, U \rangle$:

$$\begin{array}{rcl}
\overline{z} & \triangleq & \{\mathit{run}_i \ : \ i = 1,2,3,4\} \\
\Theta & \triangleq & \mathit{idle} \\
g_i & \triangleq & true \textbf{ if } \text{an action of } p_i \text{ is enabled } \textbf{else } false,\ i = 1,2,3,4 \\
sch & \triangleq & \bigvee_{i=1}^{4} (g_i \wedge (\mathit{idle} \vee \neg g_{i \oplus 1} \wedge \neg g_{\oplus 2}) \wedge \mathit{run}_i') \vee (\bigwedge_{i=1}^{4} \neg g_i) \wedge \mathit{idle}' \\
U(sch) & = & 0
\end{array}$$

Now assume that the computation times for the actions of the processes satisfy the following condition:

$$\begin{array}{ll}
l(Vote) = l(R_3^{m_i}) = l(W_3^{m_i}) = 0 & \text{for } i = 1,2,3 \\
u(W_3^{m_1}) + u(W_3^{m_2}) + u(W_3^{m_3}) \leq min\{D_{w_i}\} & \text{for } i = 1,2,3 \\
u(R_3^{m_1}) + u(R_3^{m_2}) + u(R_3^{m_3}) \leq min\{D_{r_i}\} & \text{for } i = 1,2,3 \\
u(Vote) \leq D_{vote} &
\end{array}$$

Then it can be proved using the rules in Section 9.4 that the implementation of P_3^T by the scheduler S^T on the given processor is F_2-fault-tolerantly feasible.

Intuitively, the processor actions ensure that $read$ and $write$ tasks do not arrive at the same time. Once a $write$ or a $read$ operation is issued, all the three $write$ or $read$ tasks are enabled in the three processes. The scheduler selects one process at a time to execute until all of them are executed; in total, this takes at most the sum of the computation times of the three tasks. The $Vote$ process p_4 can be ready only when the other processes are not ready.

Proof: [sketch of F_2-tolerant feasibility] Rule 3 in Section 9.5 requires that we prove that the following predicates are invariants of the F_2-implementation:

$$\begin{array}{lcll} I_R^i & \triangleq & op_i = r \Rightarrow T_{R_3^{m_i}} \leq H_{R_3^{m_i}} & \text{for } i = 1,2,3 \\ I_W^i & \triangleq & op_i = w \Rightarrow T_{W_3^{m_i}} \leq H_{W_3^{m_i}} & \text{for } i = 1,2,3 \\ I_V & \triangleq & v_1 \wedge v_2 \wedge v_3 \Rightarrow T_{Vote} \leq H_{Vote} & \end{array}$$

The proofs of these invariants are very similar. We present only a sketch of the proof for I_R^1. Let u_i be used for $u(R_3^{m_i})$, H_i for $H_{R_3^{m_i}}$, and T_i for $T_{R_3^{m_i}}$, $i = 1, 2, 3$.

In general, it may not always possible to prove an invariant Q directly from Rule $R5$ in Section 9.5. Instead, we have to use this rule to prove a stronger invariant which implies Q. To prove that I_R^1 is an invariant, prove the following invariants I_1 – I_7, the conjunction of which is an invariant and implies I_R^1:

$$I_1 \triangleq (\bigwedge_{i=1}^{3}(op_i = r \wedge \neg \mathit{run}_i)) \Rightarrow (\bigwedge_{i=1}^{3}(H_i = \mathit{now} + D_{r_i}) \wedge (T_j = \mathit{now} + u_j))$$

Notice that:

$$(7) \qquad I_1 \Rightarrow ((\bigwedge_{i=1}^{3}(op_i = r \wedge \neg \mathit{run}_i)) \Rightarrow (\bigwedge_{k \neq i \neq j \neq k}(H_i - T_i \geq u_j + u_k)))$$

Informally, I_1 is an invariant because (a) the timers H_i and T_i are respectively set with $\mathit{now} + D_{r_i}$ and $\mathit{now} + u_i$ when $op_i = r$ is changed from $false$ to $true$ and (b) there is no overhead in the scheduling and thus *now* cannot advance before one of the three ready processes is scheduled for execution.

If after a read operation is issued, the scheduler chooses process p_2 first for execution, we have the following invariant.

$$I_2 \triangleq (\bigwedge_{i=1}^{3}(op_i = r) \wedge \mathit{run}_2) \Rightarrow H_1 - T_1 \geq u_3 + T_2 - \mathit{now}$$

The proof of this invariant uses invariant I_1 and its implication 7 together with the following two facts:

1. A transition from a non-$\bigwedge_{i=1}^{3}(op_i = r) \wedge \mathit{run}_2$-state to a $\bigwedge_{i=1}^{3}(op_i = r) \wedge \mathit{run}_2$-state can only be a transition from a $\bigwedge_{i=1}^{3}(op_i = r \wedge \neg \mathit{run}_i)$-state and carried out by an scheduling action. This

scheduling action does not change *now* and the timers. Formally, let $\mathcal{N}_1$ be this action:

$$\mathcal{N}_1 \triangleq (\bigwedge_{i=1}^{3}(op_i = r \wedge \neg \mathit{run}_i)) \wedge \mathit{run}_2'$$

By I_1 and implication (7), we have:

$$\mathcal{N}_1 \Rightarrow (T_2 = \mathit{now} + u_2) \wedge (\mathit{now}, H_1 - T_1 \geq u_3 + u_2) \wedge \mathit{unchanged}(H_1, T_1, T_2)$$

Hence,

$$\mathcal{N}_1 \Rightarrow (H_1' - T_1' \geq u_3 + T_2' - \mathit{now})$$

2. The amount of time for which $\bigwedge_{i=1}^{3}(op_i = r) \wedge \mathit{run}_2$ has remained $true$ up to *now* is the time $u_2 - (T_2 - \mathit{now})$ spent on the execution of $R_3^{m_2}$ that has been added to T_1 as it has been persistent. The only action which may falsify I_2 is:

$$\begin{array}{lll} \mathcal{N}_2 & \triangleq & (op_1 = r) \wedge (op_2 = r) \wedge (op_3 = r) \wedge \mathit{run}_2 \\ & \wedge & (\mathit{now}' > \mathit{now}) \wedge (T_1' = T_1 + (\mathit{now}' - \mathit{now})) \\ & \wedge & (T_2' = T_2) \wedge (H_1' = H_1) \end{array}$$

where we ignore the changes in other variables which are irrelevant to I_2. Clearly $I_2 \wedge \mathcal{N}_2 \Rightarrow I_2'$ as:

$$\begin{array}{lll} H_1' - T_1' & = & H_1 - T_1 - \mathit{now}' + \mathit{now} \\ & \geq & u_3 + T_2 - \mathit{now} - \mathit{now}' + \mathit{now} \\ & = & u_3 + T_2' - \mathit{now}' \end{array}$$

Similar to I_2, we have the following invariant if the scheduler chooses p_3 first for execution:

$$I_3 \triangleq (\bigwedge_{i=1}^{3}(op_i = r) \wedge \mathit{run}_3) \Rightarrow H_1 - T_1 \geq u_2 + T_3 - \mathit{now}$$

If the scheduler chooses p_1 for execution first, then:

$$I_4 \triangleq (\bigwedge_{i=1}^{3}(op_i = r) \wedge \mathit{run}_1) \Rightarrow H_1 - T_1 \geq u_1 + u_3$$

These four invariants consider the cases when none of the three processes has completed the issued read operation. We have the following three invariants about the cases when one of or both of p_2 and p_3 have completed the operation.

If p_2 has completed the operation, we have the invariant:

$$I_5 \triangleq (op_1 = r) \land (op_2 \neq r) \land (op_3 = r) \land \neg \mathit{run}_1 \Rightarrow H_1 - T_1 \geq T_3 - \mathit{now}$$

This characterises the fact that the time spent on the whole execution of $R_3^{m_2}$ and on the partial execution of $R_3^{m_3}$ has been added to T_1. The proof of this invariant uses I_2. Similarly, if p_2 has completed the operation we have the invariant:

$$I_6 \triangleq (op_1 = r) \land (op_2 = r) \land (op_3 \neq r) \land \neg \mathit{run}_1 \Rightarrow H_1 - T_1 \geq T_2 - \mathit{now}$$

Finally, we have the invariant:

$$I_7 \triangleq (op_1 = r) \land ((op_2 \neq r) \lor (op_3 \neq r)) \Rightarrow H_1 - T_1 \geq 0$$

This characterises the fact that the time spent on the execution of one or both of $R_3^{m_2}$ and $R_3^{m_3}$ has been added to T_1. ♡

The nondeterministic scheduler can be refined to a deterministic one by assigning priorities to the processes. For example, let process p_i have a higher priority than p_j if $i < j$. Modify the action sch of the scheduler into sch_1 such that the process with the highest priority among the ready processes is scheduled for execution but no pre-emption is allowed:

$$\begin{array}{lcll}
\mathit{sch}_1 & \triangleq & & \mathit{idle} \land g_1 \land \mathit{run}_1' \\
& & \lor & \mathit{idle} \land \neg g_1 \land g_2 \land \mathit{run}_2' \\
& & \lor & \mathit{idle} \land \neg g_1 \land \neg g_2 \land g_3 \land \mathit{run}_3' \\
& & \lor & \mathit{idle} \land \neg g_1 \land \neg g_2 \land \neg g_3 \land g_4 \land \mathit{run}_4' \\
& & \lor & \bigvee_{i=1}^{4} (\mathit{run}_i \land \neg g_i \land \mathit{idle}')
\end{array}$$

Then the modified scheduler also gives a feasible F_2-tolerant implementation of P_3^T on the given processor, as the new action sch_1 action implies the old sch.

♣

9.9 Fixed priority scheduling with pre-emption

The techniques presented in the previous subsections can be used to produce results similar to those obtained using scheduling theory. We demonstrate this by proving the feasibility condition given in [14] for implementing a set of independent tasks using fixed priority scheduling with pre-emption.

Consider an open system $O = (E, P)$ where program P consists of n independent processes (or tasks) which are represented by the atomic actions $\tau_1, \ldots, \tau_n$. The environment E is used to represent the actions of releasing (or invoking, or activating) the tasks periodically. In general, these actions may be clock events or external events to which the processes need to respond. Let ρ_i be the period of τ_i, for $i = 1, \ldots, n$.

9.10 Specification of the program

To specify the system in TLA, let *inv*$_i$ and *com*$_i$ be integer variables representing the number of invocations and completions of each task i. Then the specification of the real-time system $O^T = (E^T, P^T)$ can be given as: for $i = 1, \ldots, n$

$$
\begin{array}{lcll}
\Theta & \triangleq & (0 \leq \mathit{inv}_i \leq 1) \wedge (\mathit{com}_i = 0) & \\
\alpha & \triangleq & \mathit{inv}'_i = \mathit{inv}_i + 1 & \text{action of } E \text{ for task invocation} \\
\tau_i & \triangleq & \mathit{inv}_i > \mathit{com}_i \wedge \mathit{com}'_i = \mathit{com}_i + 1 & \text{action of } P \text{ for task completion} \\
\nu & \triangleq & \vee_{i=1}^{n} (\mathit{com}'_i = \mathit{com}_i + 1) & \\
L(\alpha_i) & = & U(\alpha_i) = \rho_i & \text{period of invocation} \\
L(\tau_i) & = & 0 \text{ and } U(\tau_i) = D_i & \text{deadline of task}
\end{array}
$$

A basic (functional) requirement for the system is that each invocation of a task is completed before its next invocation, i.e.

$$\Phi(E^T) \wedge \Phi(P^T) \Rightarrow \bigwedge_{i=1}^{i=n} \Box(\mathit{inv}_i \geq \mathit{com}_i \geq \mathit{inv}_i - 1)$$

From the rules for proving an invariant in TLA, this implication holds if $D_i < \rho_i$. It must now be shown that an implementation of the program P^T on a uniprocessor system is feasible.

9.11 Specification of the scheduling policy

Let the system be implemented on a single processor using a pre-emptive, fixed-priority scheduler[6]; assume that there is no scheduling overhead. Let τ_i have a higher priority than τ_j if $i < j$. Let g_i denote the enabling condition of task τ_i, and hr_i assert that τ_i has the highest priority among the current enabled (or ready) tasks:

$$g_i \triangleq \mathit{inv}_i > \mathit{com}_i \qquad hr_i \triangleq g_i \wedge \forall j < i. \neg g_j$$

[6]Specifications of various scheduling policies can be found in [60].

Then the scheduler, denoted by $S^T = \langle S, L, U \rangle$, can be specified as follows:

$$\begin{array}{rcll} \mathit{sch}_i & \triangleq & \mathit{idle} \wedge hr_i \wedge \mathit{run}'_i & \text{higher task runs first} \\ & \vee & \exists j \neq i.(\mathit{run}_j \wedge hr_i \wedge \mathit{run}'_i \wedge \neg \mathit{run}'_j) & \text{higher task pre-empts lower task} \\ \mathcal{N}_S & = & \bigvee_{i=1}^{n} \mathit{sch}_i & \\ U(\mathit{sch}_i) & = & L(\mathit{sch}_i) = 0 & \text{no overhead} \end{array}$$

According to S^T, at any time at most one process is running on the processor:

$$\mathit{Valid} \triangleq \Box(i \neq j \Rightarrow \neg(\mathit{run}_i \wedge \mathit{run}_j))$$

9.12 Feasibility

Let the computation time for each task τ_i be in the interval $[0, C_i]$, i.e. $l(\tau_i) = 0$ and $u(\tau_i) = C_i$. Assume ρ_i, D_i and C_i are non-negative integers for $i = 1, \ldots, n$. The worst-case response time (or completion time) R_i for each task τ_i can be defined as a recursive equation [38]. We shall instead use the equivalent recurrence relation defined in [14]. The $(n+1)$th response time $R_i^{(n+1)}$ for process i is:

$$(8) \qquad R_i^{(n+1)} = C_i + \sum_{j=1}^{i-1} \lceil \frac{R_i^{(n)}}{\rho_j} \rceil \times C_j$$

If $R_i^{(0)}$ is initially set C_i, and:

$$R_i = \lim_{n \to \infty} R_i^{(n)}$$

scheduling theory shows that:

> the implementation of the program by the scheduler on the given processor is *feasible* iff $R_i \leq D_i$, for $i = 1, \ldots, n$.

This condition can be shown to be necessary by finding an execution in which a task misses its deadline if the condition does not hold. However, to prove formally that the condition is sufficient, we need to prove the following refinement.

Theorem 1 *For the given program, $O^T = (E^T, P^T)$, the scheduler, S^T, and the processor*

$$\mathcal{I}(O^T) \sqsubseteq O^T$$

provided $R_i \leq D_i$ for $i := 1, \ldots, n$.

By Implication (6) in Section 9.7, this is equivalent to showing that the following holds:

$$\text{(9)} \quad \Pi(E^T) \wedge \Pi(S^T) \wedge \Pi(\mathcal{G}(P^T)) \wedge D(\textit{dummies}) \Rightarrow \widetilde{\Pi(P^T)}$$

where $D(\textit{dummies})$ and refinement mapping are as defined in Section 5.1.

Before proving (9), let us discuss how the persistent timer T_{τ_i} is used to predict the completion time of an invocation of task τ_i by considering its first invocation.

As a special case, consider any time *now* before the completion of the *first* invocation of task τ_i (i.e. when $\textit{com}_i = 0$ and $\textit{inv}_i > 0$). Assume all tasks τ_j, $j = 1, \ldots, i-1$, with higher priorities than τ_i have met their deadlines so far. Then, in the worst case, when all tasks τ_j use C_j units of computation time, the time spent up to *now* on executing higher priority processes is:

$$\text{(10)} \quad \textit{Comp}(i, \textit{now}) \triangleq \sum_{j=1}^{i-1} \textit{com}_j \times C_j + \sum_{j=1}^{i-1} (\textit{inv}_j - \textit{com}_j) \times (C_j - (T_{\tau_j} - \textit{now}))$$

where $C_j - (T_{\tau_j} - \textit{now})$ is the time spent so far on the last invocation of τ_j. Thus (10) becomes:

$$\text{(11)} \quad \textit{Comp}(i, \textit{now}) = \sum_{j=1}^{i-1} \textit{inv}_j \times C_j - \sum_{j=1}^{i-1} (\textit{inv}_j - \textit{com}_j) \times (T_{\tau_j} - \textit{now})$$

Assume δ is the time already spent on τ_i up to *now*. Then:

$$\textit{now} = \textit{Comp}(i, \textit{now}) + \delta$$

As T_{τ_i} has been persistent during the time when tasks of higher priorities are being executed, we have

$$T_{\tau_i} = \textit{Comp}(i, \textit{now}) + C_i$$

Thus, $T_{\tau_i} = \mathit{now} + (C_i - \delta)$ *predicts* that the cumulative time needed to complete τ_i after *now* will not exceed $C_i - \delta$; this time may be divided into smaller units whose sum is T_{τ_i}. For the first invocation of τ_i to be completed before its deadline, T_{τ_i} should never exceed H_{τ_i} (which is always equal to D_i before the completion of τ_i).

Thus, we need to prove that the left hand side (or LHS) of Implication (9) has the following predicate as an invariant:

$$(\mathit{com}_i = 0 \wedge \mathit{inv}_i > \mathit{com}_i) \Rightarrow T_{\tau_i} \leq C_i + \mathit{Comp}(i, \mathit{now})$$

In general, at any time before an invocation of τ_i is completed, $H_{\tau_i} - D_i$ records the time t_0 (i.e. the value of *now* at that time) of the current invocation of τ_i: at that time H_{τ_i} was $t_0 + D_i$ and it has remained unchanged as τ has not been completed. The definition of the longest possible time, $\mathit{Comp}(i, \mathit{now})$, spent on executing tasks with priorities higher than that of τ_i's defined by Equation (11) can be generalised as:

$$\begin{aligned}\mathit{Comp}(i, \mathit{now}) \quad \triangleq \quad & \sum_{j=1}^{i-1} \lceil \frac{\mathit{inv}_j \times \rho_j - (H_{\tau_i} - D_i)}{\rho_j} \rceil \times C_j \\ & - \sum_{j=1}^{i-1} (\mathit{inv}_j - \mathit{com}_j) \times (T_{\tau_j} - \mathit{now})\end{aligned}$$

This leads to the following lemma which implies Theorem 1.

Lemma 1 *LHS(9) has the following invariants: for* $i = 1, \ldots, n$

$$\begin{aligned}
I_{1i} \quad &\triangleq \quad (\mathit{com}_i < \mathit{inv}_i) \Rightarrow T_{\tau_i} \leq C_i + \mathit{Comp}(i, \mathit{now}) + (H_{\tau_i} - D_i) \\
I_{2i} \quad &\triangleq \quad (\mathit{com}_i < \mathit{inv}_i) \Rightarrow \mathit{Comp}(i, \mathit{now}) \leq \sum_{j=1}^{i-1} \lceil (\mathit{now} - (H_{\tau_i} - D_i))/\rho_j \rceil \times C_j \\
I_{3i} \quad &\triangleq \quad (\mathit{com}_i < \mathit{inv}_i) \Rightarrow C_i + \mathit{Comp}(i, \mathit{now}) \leq R_i \\
I_{4i} \quad &\triangleq \quad \mathit{inv}_i - 1 \leq \mathit{com}_i \leq \mathit{inv}_i
\end{aligned}$$

Proof: [of Lemma 1] The proof follows the general routine of proving invariants by showing that each of the I's holds initially and is preserved by each allowed state transition in the program.

It is easy to check that these invariants hold for $i = 1$. Assume that they hold for some $i - 1$, where $i \geq 1$. We prove they hold for i.

Take the case when $H_{\tau_i} = D_i$ for the first invocation of τ_i, i.e. the execution of the first invocation of τ_i. (The proof of the general case is very similar.)

For the special case, the lemma is rewritten as follows:

$$\begin{array}{lcl}
I_{1i} & \triangleq & (\mathit{com}_i = 0) \wedge (\mathit{inv}_i > 0) \Rightarrow T_{\tau_i} \leq C_i + \mathit{Comp}(i, \mathit{now}) \\
I_{2i} & \triangleq & (\mathit{com}_i = 0) \wedge (\mathit{inv}_i > 0) \Rightarrow \mathit{Comp}(i, \mathit{now}) \leq \sum_{j=1}^{i-1} \lceil \mathit{now}/\rho_j \rceil \times C_j \\
I_{3i} & \triangleq & (\mathit{com}_i = 0) \wedge (\mathit{inv}_i > 0) \Rightarrow C_i + \mathit{Comp}(i, \mathit{now}) \leq R_i \\
I_{4i} & \triangleq & \mathit{inv}_i - 1 \leq \mathit{com}_i \leq \mathit{inv}_i
\end{array}$$

where:

$$\mathit{Comp}(i, \mathit{now}) = \sum_{j=1}^{i-1} \mathit{inv}_j \times C_j - \sum_{j=1}^{i-1} (\mathit{inv}_j - \mathit{com}_j) \times (T_{\tau_j} - \mathit{now})$$

Initially, I_{1i} holds as $T_{\tau_i} = C_i$. We analyze all the possible state transitions allowed by $LHS(9)$ which may change the states of variables occurring in $\mathit{Comp}(i, \mathit{now})$.

Case 1: For $j = 1, \ldots, i-1$, let

$$A_{1j} \triangleq \mathit{com}_i = 0 \wedge \mathit{inv}'_j = \mathit{inv}_j + 1$$

In this case, I'_{1i} is

$$\mathit{com}'_i = 0 \Rightarrow T_{\tau_i} \leq C_i + \mathit{Comp}(i, \mathit{now}) + C_j - (T_{\tau_j} - \mathit{now})$$

It is easy to prove that $LHS(9) \Rightarrow \Box(C_j - (T_{\tau_j} - \mathit{now} > 0))$. Thus,

$$I_{1i} \wedge A_{1j} \Rightarrow I'_{1i}$$

Case 2: For $j = 1, \ldots, i-1$, consider

$$A_{2j} \quad \triangleq \quad \mathit{com}_i = 0 \wedge \mathit{com}'_j = \mathit{com}_j + 1 \wedge (T'_{\tau_j} = \mathit{now} + C_j)$$

By the induction assumption that $\Box(\mathit{inv}_j - 1 \leq \mathit{com}_j \leq \mathit{inv}_j)$, we know that I'_{1i} is equal to

$$\begin{array}{lcl}
\mathit{com}'_i = 0 & \Rightarrow & T_{\tau_i} \leq C_i + \sum_{j=1}^{i-1} \mathit{inv}_j \times C_j \\
& & - \sum_{i > k \neq j} (\mathit{inv}_k - \mathit{com}_k) \times (T_{\tau_k} - \mathit{now})
\end{array}$$

Thus, $I_{1i} \wedge A_{2j} \Rightarrow I'_{1i}$ holds.

Case 3: For $j = 1, \ldots, i-1$, define

$$\begin{array}{rcl} A_{3j} & \triangleq & \mathit{com}_i = 0 \wedge (\mathit{com}_j < \mathit{inv}_j) \wedge \mathit{run}_j \\ & \wedge & (\mathit{now}' > \mathit{now}) \wedge T'_{\tau_i} = T_{\tau_i} + (\mathit{now}' - \mathit{now}) \wedge \varphi \end{array}$$

where

$$\varphi \triangleq \bigwedge_{i>k\neq j} (\mathit{com}_k < \mathit{inv}_k \Leftrightarrow (T'_{\tau_k} = T_{\tau_k} + (\mathit{now}' - \mathit{now})))$$

Note that $(\mathit{inv}_k - \mathit{com}_k) = 0$ iff $\neg(\mathit{com}_k < \mathit{inv}_k)$ by the induction assumption for $k < i$. Thus, I'_{1i} becomes

$$\begin{array}{rcl} \mathit{com}_i = 0 & \Rightarrow & T_{\tau_i} + (\mathit{now}' - \mathit{now}) \\ & \leq & C_i + \sum_{k=1}^{i-1} \mathit{inv}_k \times C_k - \\ & & \sum_{i>k\neq j} (\mathit{inv}_k - \mathit{com}_k) \times (T_{\tau_k} - \mathit{now}) - (T_{\tau_j} - \mathit{now}') \end{array}$$

This is the same as

$$\begin{array}{rcl} \mathit{com}_i = 0 \Rightarrow T_{\tau_i} & \leq & C_i + \sum_{k=1}^{i-1} \mathit{inv}_k \times C_k - \\ & & \sum_{k=1}^{i-1} (\mathit{inv}_k - \mathit{com}_k) \times (T_{\tau_k} - \mathit{now}) \end{array}$$

Thus, $I_{1i} \wedge A_{3j} \Rightarrow I'_{1i}$ holds.

Case 4: Finally consider:

$$A_4 \triangleq (\mathit{com}_i = 0) \wedge \mathit{run}_i \wedge \varphi \wedge (\mathit{now}' > \mathit{now})$$

where φ is defined as in **Case 3**, except for j being taken into account. Then, the same argument as in **Case 3** leads to I'_{1i} becoming:

$$\begin{array}{rcl} \mathit{com}_i = 0 \Rightarrow T_{\tau_i} & \leq & C_i + \sum_{j=1}^{i-1} \mathit{inv}_j \times C_j - \\ & & \sum_{j=1}^{i-1} (\mathit{inv}_j - \mathit{com}_j) \times (T_{\tau_j} - \mathit{now}) \end{array}$$

And $I_{1i} \wedge A_4 \Rightarrow I'_{1i}$ holds.

These four cases prove Invariant I_{1i}. The proof for I_{2i} follows from the facts:

$$\mathit{now} = m \times \rho_j \text{ iff } \mathit{inv}_j \leq (m+1) \text{ and } \mathit{inv}_j - \mathit{com}_j = 1 \text{ and } T_{\tau_j} = \mathit{now} + C_j$$

and

$$\lceil now/\rho_j \rceil \geq inv_j \text{ if } now = m \times \rho_j + t_0, \text{ where } 0 < t_0 < \rho_j$$

To prove that I_{3i} is an invariant, note that $MaxTime(T_{\tau_i})$ requires:

(12) $now' \leq T_{\tau_i} \leq C_i + Comp(i, now)$

For any allowed transition A, assume that $C_i + Comp(i, now) \leq R_i \wedge A$. Then by I_{2i} and the inequation (12):

$$\begin{array}{rcll} Comp(i, now') + C_i & \leq & \sum_{j=1}^{i-1} \lceil now'/\rho_j \rceil \times C_j + C_i & I_{2i} \text{ of the lemma} \\ & \leq & \sum_{j=1}^{i-1} \lceil (C_i + Comp(i, now))/\rho_j \rceil + C_i & \text{inequation (12)} \\ & \leq & \sum_{j=1}^{i-1} \lceil R_i/\rho_j \rceil \times C_j + C_i R_i & \text{Definition of } R_i \end{array}$$

The general cases for I_{1i}, I_{2i} and I_{3i} can be proved in the same way. From the assumption that $R_i \leq D_i$, these three cases together guarantee that $\Box(T_{\tau_i} \leq H_{\tau_i})$ and thus the deadline for the task is always met. This ensures I_{4i} holds. Notice that I_{4i} is not used in the proof, though I_{4j}, for $j = 1, \ldots, i-1$, are used as the induction assumption. Therefore, we have proved the Lemma. ♡
The proof of Theorem 1 follows Rule R3 in Section 9.4 in a straightforward way from this Lemma.

9.13 Discussion

The example in Section 9.9 deals with independent periodic tasks with fixed priorities. The method in scheduling theory used for these tasks has been extended to deal with communicating tasks. For example, tasks may communicate with each other asynchronously through a *protected shared object* (PSO) [14]. These tasks may be periodic or *sporadic*. For a scheduler with *ceiling priorities*, the worst response time R_i for a task τ_i can be calculated by the recurrence relation:

$$R_i^{(k+1)} = B_i + C_i + \sum_{j=1}^{i-1} \lceil \frac{R_i^{(k)}}{\rho_j} \rceil \times C_j$$

where B_i is the worst blocking time for τ_i by a task of lower priority, and ρ_j is minimum inter-arrival time of task τ_j (which is the period of τ_j if τ_j is periodic).

In the feasibility analysis of fault-tolerant real-time tasks [12], the recurrence relation for the worst response time R_i for a task τ_i has been extended to deal with fault-tolerant tasks: by re-execution of the affected task, by forward recovery, by recovery blocks, by checkpointing and backward recovery. In the case of fault-tolerance by re-execution, the response time R_i for a task τ_i can be calculated by the recurrence relation:

$$R_i^{(k+1)} = B_i + C_i + \sum_{j=1}^{i-1} \lceil \frac{R_i^{(k)}}{\rho_j} \rceil \times C_j + \lceil \frac{R_i^{(k)}}{F_j} \rceil \times max\{C_j : 1 \leq j \leq i\}$$

where F_j is the minimum time between two occurrences of faults.

The formal method for scheduling analysis presented here can be applied to communicating, fault-tolerant tasks. This allows us to combine this work with our previous work on fault-tolerance and real-time [50, 53, 54, 55, 56], which formally treat re-execution, forward recovery, recovery blocks, and checkpointing and backward recovery, and provide a means of formally dealing with real-time program refinement, fault-tolerance and schedulability in a single and consistent framework.

10 Related Work

There have been a number of other approaches to formalising real-time scheduling. Using the Duration Calculus [86], Zhou Chaochen et al [85, 84] have also separately specified a scheduler and a scheduled program. However, the Duration Calculus does not at present have powerful verification tools for proving program refinement. It would be useful to unify the theories of Linear Temporal Logics and Duration Calculus for the specification and analysis of real-time systems. Work in this direction is making slow progress [61, 18, 19].

[63] describes a case study using a ‘scheduling-oriented model for real-time systems’ called TAM. The work of [26] extends Back’s action systems [9] with timing and priorities and uses the **Z** notation for specification and refinement. The models used there appear to be more complicated than is necessary. For example, priorities and scheduling can be defined using only simple state variables and standard actions, as we have shown here, and complex models and structures are not needed.

Using timed CCS, [35, 36] deals with dynamic scheduling in the presence of faults by modelling resources and schedulers as processes. This serves well as a model but event-based process algebras tend to have a very different syntax to most traditional programming languages; it is possible to consider extensions to this work which make use of persistent timers and this would enable pre-emption to be modelled. The similar approach should also be applicable to the framework of CSP presented in Chapter 2 of this volume. For example, [72] deals with physical faults and verification of fault-tolerance by using the notation of CSP. As in our earlier work [60], these approaches use volatile time bounds (either explicitly or implicitly) for both program verification and scheduling analysis. When dealing with pre-emption (interruption) in real-time scheduling, the use of volatile time bounds requires a scheduled action to be *explicitly* divided into smaller actions (or steps) between whose execution pre-emption can occur. The atomicity of the original action has to be preserved and this requires the introduction of auxiliary internal variables. The feasibility of the implementation is established by reasoning about this 'step-level' program. The use of these devices makes it difficult to reason about and make formal use of the methods and results from scheduling theory, especially as this does in fact make (informal) use of the accumulated execution time of tasks.

Another approach to the verification of schedulability uses algorithms for computing *quantitative* information of an implementation, such as the lower bound and upper bound on the delay between two (or two sets of) states [16, 15]. The quantitative information is then used to determine the feasibility of the implementation and to verify other timing properties using *symbolic model checking* techniques [66]. There are some significant differences between that work and what we have described:

1. The algorithms and the model-checking procedures described in [16] work effectively with a discrete time domain and a finite-state system; in contrast, in our analysis time is modelled by the reals and systems may have a finite or an infinite set of states.
2. Our framework allows program development through refinement to be integrated with scheduling theory so that the methods and results from the latter can be formal interpreted, verified and used correctly. [16] uses a scheduling algorithm to obtain an implementation and then tests for schedulability. There is no verification of whether a theorem in scheduling theory is valid for the program model used (compare this with Section 9.9). In fact, application of Theorem 1 and the recurrence relation 8 to the Aircraft Control System example of [16, 15] leads directly to the same feasibility conclusion obtained there.
3. Compared with the work in [16, 15] which concentrates on timing aspects, this treatment deals with the much wider range of inter-related issues of concurrency, timing, fault-tolerance and schedulability, as well as refinement techniques for fault-tolerant and real-time programs.

In general, model checking techniques are especially effective and necessary in many safety-critical applications (please see Chapter 7 David Deharbe on Model Checking). However, their general applicability has been restricted by questions of undecidability [6] and by complexity issues [5], especially for systems using a continuous time domain. These problems are very much more serious when both fault-tolerance and real-time have to be considered. Model-checking and the more general verification methods used here are complementary and neither can be totally replaced by the other. We recognize a role for model-checking as a decision procedure in a proof-checker, to be applied when possible.

It is usually impossible to give an exact prediction for the occurrence of faults in a program execution, or to achieve one hundred per cent fault-tolerance. Therefore, fault-tolerance is often addressed with the concepts of dependability and reliability. The occurrence of faults is associated with a probability distribution and verification of fault-tolerance is thus related to the calculation of reliability based on the probability distribution. There is no much work on formal models to support effective reasoning about reliability. We believe interesting work can be done by combining the model in this chapter with that of Chapter 3 by Morgan on Probability. The idea of Unifying Theories of Programm in [33] will be very useful for this combination.

11 Conclusions

Formal development and verification of a real-time program requires a logical structure in which functional and timing properties of the program can be specified and reasoned about. In many practical cases, such programs are executed under a scheduler whose actions control the program's execution and thus its timing properties. A program is also often executed on a failure-prone system and thus fault-tolerance is needed. However, fault-tolerance and schedulability affect each other and they both affect the functionality and timing of the program. This chapter presents a framework which we believe is suitable for a coherent understanding of the relationship between theories of concurrency, real-time, fault-tolerance and schedulablity analysis; and for formal and systematic development of safety and/or timing critical computer systems.

Scheduling theory provides powerful techniques for determining the timing properties of a restricted class of real-time programs; however, it does not provide any means of verifying functional properties. Such methods must be augmented by more traditional program verification techniques, but these use a different analytical framework, making it hard to relate the results in a rigorous way. This is particularly important when mechanised verification is to be performed and the program's properties certified, as is necessary in many safety-critical applications.

In a separate paper [60], we showed how the schedulability of a real-time program could be established using techniques very similar to those used here. An important observation that can be made about that work is that to simplify verification it is useful to reduce the number of actions by specifying them at *as high* a level as possible. However, for accurate verification of timing properties it is necessary to have a fine level of granularity in the time bounds for each action and each deadline: this requires specifying actions at *as low* a level as possible, so that pre-emption can be precisely modelled and the timing properties related to those obtained from scheduling theory.

We address this issue in this chapter by providing two kinds of timers: volatile timers that record times for which actions are continuously enabled, and persistent timers that sum the duration for which actions are executed. The use of persistent timers allows the timing effects of lower-level actions, like pre-emption, to be considered abstractly and at a higher-level. It no longer matters exactly when an action is pre-empted: what is important is the time for which it executed before pre-emption and the time for which it is pre-empted. Thus an action may be pre-empted a number of times and still make use of a single timer to record its timing properties.

The use of two kinds of timers solves a problem that has been the cause of a major restriction in the application of formal verification methods in the validation of real-time programs. It makes it feasible to use automated verification for such programs at the specification level, allowing timing properties to be considered well before the details of the implementation have been finalised. Naturally, once the implementation is complete, scheduling analysis will still be required to validate and provide independent certification of the timing properties.

The method presented in this chapter is independent of a programming language. Also, both the program and the scheduler specifications can be refined, with feasibility and correctness being preserved at each step. This has the great advantage that proving feasibility does not first require the code of the program to be developed.

There are many advantages to using a single, consistent treatment of fault-tolerance, timing and schedulability. Not only does it allow a unified view to be taken of the functional and non-functional properties of programs and a simple transformational method to be used to combine these properties, it also makes it possible to use a uniform method of verification. Verification of schedulability within a proof framework will inevitably be more cumbersome than using a simple schedulability test from scheduling theory. However, the use of a common framework means that during formal verification, the test for schedulability can be defined as a theorem whose verification is not actually done within the proof theory but instead by invoking an oracle or decision procedure which uses scheduling theory for rapid analysis.

The plan of our future work includes the combination of the techniques presented in this chapter with those developed in our recent work on object-oriented and component based systems [29, 51]. We hope such a combination will lead to a multi-view and multi-notational framework for modelling, design, analysis and verification of real-time and fault-tolerant systems at different levels of abstraction. It will also support transformational, incremental and iterative development [52, 83] aided with transformation and verification tools [48, 62, 82, 1].

12 Exercises

1. Relating the notation of this chapter with other formalisms.
   (a) Specify sequential composition as TLA action?

   $$\tau_1;\ \tau_2$$

   where each action is treated as an atomic action.

   What about when the whole composed $< \tau_1;\ \tau_2 >$ is treated as an atomic action?

   (b) Model the conditional choice as a TLA action

   $$\mathbf{if}\ b_1\ \mathbf{then}\ \tau_1\ \mathbf{else}\ \tau_2$$

   (c) Define Hoare triple $\{p\}\tau\{q\}$ as a TLA action.

   (d) Define the Morgan's specification statement $w : [p;\ q]$ as a TLA action.

   (e) Define the non-deterministic choice $\tau_1 \sqcap \tau_2$ in TLA.

   (f) Understand how does the TLA notation unify the semantics of deterministic choice and non-deterministic choice.

2. Consider the problem of Dinning Philosophers. Assume there are five philosophers, p_i, $i = 1, \ldots, 5$, and five chopsticks, c_i, $i = 1, \ldots, 5$, that are placed in five positions a dinning table . The life of each philosopher is repeatedly *thinking* and *eating*. Assume that initially, all philosophers are thinking. After thinking, a philosopher p_i becomes hungry and want to eat. To eat, he has to come to the position (i.e. chair) at the dinning table reserved form him and gets the chopstick c_i on his left and then the one, c_{i+1}, on his right. A philosopher cannot start to eat before he gets both sticks. After eating, a philosopher puts down both chopsticks and goes back to think.

(a) Write a TLA specification of the problem of the dinning philosophers.

(b) Specify in TLA that the fairness condition that an eating philosopher will eventually put down the chopsticks.

(c) Specify the liveness property that no philosopher can be starved.

(d) Does your specification for part (a) satisfies the liveness property under the fairness condition (deadlock freedom)?

(e) Suggest solutions to fix deadlock problem in the specification of part (a), and write the TLA specifications for these solutions.

3. Consider the Gas Burner example in [79, 42]. This case study formulates the safety requirement of a gas burner in terms of a variable *Leak* denoting an undesirable but unavoidable state which represents the presence of unlit gas.

   For safety, '*gas must never leak for more than 4 seconds in any period of at most 30 seconds*'. This is specified by the bounded critical duration property:

   To meet the requirement *Req*, two design decisions are made:

   Des-1 any occurrence of leak must be stopped within 4 seconds, and

   Des-2 two occurrences of leaks must be separated by a period of 26 seconds in which the burner does not leak; in other words, a *Leak* is stopped it may not reoccur within 26 seconds.

   (a) Write TLA specifications for the above design decisions.

   (b) Reason within TLA that the above two design decisions are met by the timed transition system defined below:

$$\begin{array}{lcll} GB_1 & = & \langle & \Theta_1 : true & \\ & & & \tau_1 : Leak \wedge \neg Leak' & [0, 4] \\ & & & \tau_2 : \neg Leak \wedge Leak' & [26, \infty) \\ & & \rangle & & \end{array}$$

   (c) After the initial design, *GB*$_1$ can be refined. For example, the transition system *GB*$_2$ in Figure 3 is a refinement of *GB*$_1$.

   GB$_2$ has the following *phases*:

   Idle: Await heat request with no gas and no ignition. It enters **Purge** within e time units on heat request. The parameter e in this example is the system wide upper bound for *reactions*.

   Purge: Pauses for 30 seconds and then enters **Ignite1** within e time units.

   Ignite1: Turns on ignition and gas and after one second exits within e to **Ignite2**.

   Ignite2: Monitors the flame, if it is sensed within one second **Burn** is entered, otherwise it returns to **Idle** within e while turning the gas off.

   Burn: Ignition is switched off, but gas is still on. The **Burn** phase is stable until heat request goes off. Gas is then turned off and **Idle** is entered within e.

   This refinement uses a simple error recovery: return to **Idle** from **Ignite2**. We assume no flame failure in the **Burn** phase. Therefore, in this implementation, *Leak* can only occur in the **Ignite1** and **Ignite2** phases.

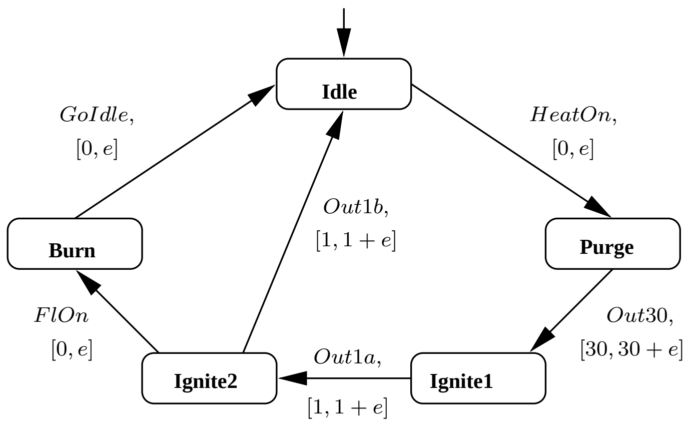

Figure 3: A refinement of GB_1

(d) Formalize in TLA the full and canonical specification of GB_2, and decide the constant e such that GB_2 also meets the two design decisions for GB_1.

4. A Project for Self Study: Apply the notation and techniques to the development of the realtime mine pump system described in Section 1 (please see [13]). Then extend the solution to deal with fault-tolerance (please see Chapter 8 in [13]).

References

[1] Mastercraft. Tata Consultancy Services. http://www. tata-mastercraft.com.

[2] M. Abadi and L. Lamport. The existence of refinement mapping. *Theoretical Computer Science*, 83(2):253–284, 1991.

[3] M. Abadi and L. Lamport. An old-fashioned recipe for real-time. In J.W. de Bakker, C. Huizing, W.P. de Rover, and G. Rozenberg, editors, *Real-Time: Theory in Practice, Lecture Notes in Computer Science 600*. Springer-Verlag, The Netherlands, 1992.

[4] A. Alur, C. Courcoubetis, and D. Dill. Model-checking for real-time systems. In *Proceedings of the 5th IEEE Symposium on Logic in Computer Science*, pages 414–425, Philadelphia, USA, 1990. IEEE Computer Society Press.

[5] A. Alur and T.A. Henzinger. Real-time logics: complexity and expressiveness. In *Proceedings of the 5th Annual Symposium on Logic in Computer Science*, pages 390–401, Philadelphia, USA, 1990. IEEE Computer Society.

[6] R. Alur and D.L. Dill. Automata for modelling real-time systems. In M.S. Paterson, editor, *ICALP 90: Automata, Languages and Programming, Lecture Notes in Computer Science 443*, pages 322–335. Springer-Verlag, Warwick, U.K., 1990.

[7] N. Audsley, A. Burns, M. Richardson, K. Tindell, and A. Wellings. Applying new scheduling theory to static priority pre-emptive scheduling. Technical Report RTRG/92/120, Department of Computer Science, University of York, 1992.

[8] A. Avižienis. Fault-tolerant systems. *IEEE Transactions on Software Engineering*, C-25(12):1304–1312, December 1976.

[9] R.J.R. Back. A calculus of refinements for program derivations. *Acta Informatica*, 25:593–624, 1988.

[10] N.S. Bjørner, Z. Manna, H.B. Sipma, and T.E. Uribe. Deductive verification of real-time systems using STeP. In M. Bertran and T. Rus, editors, *Proceedings of Transformation-Based Reactive Systems Development, ARTS'97, Lecture Notes in Computer Science 1231*, pages 21–43. Springer, Palma, Mallorca, Spain, 1997.

[11] J. Burch, E. Clarke, D. Dill, L. Hwang, and K. McMillan. Symbolic model checking: 10^20 states and beyond. In *Proceedings of the 5th IEEE Symposium on Logic in Computer Science*, pages 428–439, Philadelphia, USA, 1990. IEEE Computer Society Press.

[12] A. Burns, R. Davis, and S. Punnekkat. Feasibility analysis of fault-tolerant real-time tasks. Technical report, Department of Computer Science, University of York, 1995.

[13] A. Burns, J. Hooman, M. Jospeh, Z. Liu, K. Ramamritham, H. Schepers, S. Schneider, and A.J. Wellings. *Real-time systems: Specification, Verification and analysis*. Prentice Hall International, 1996.

[14] A. Burns and A. Wellings. Advanced fixed priority scheduling. In M. Joseph, editor, *Real-Time Systems: Specification, Verification and Analysis*, pages 32–65. Prentice Hall, London, 1996.

[15] S. Campos and E. Clarke. The Verus language: representing time efficiently with BDDs. In M. Bertran and T. Rus, editors, *Proceedings of Transformation-Based Reactive Systems Development, ARTS'97, Lecture Notes in Computer Science 1231*, pages 64–78. Springer, Palma, Mallorca, Spain, 1997.

[16] S. Campos, E.M. Clarke, W. Marrero, M. Minea, and H. Hiraishi. Computing quantitative characteristics of finite-state real-time systems. In *Proceedings of IEEE Real-time Systems Symposium*, San Juan, Puerto Rico, USA, 1994. IEEE Computer Society Press.

[17] K.M. Chandy and J. Misra. *Parallel Program Design: A Foundation*. Addison-Wesley Publishing Company, New York, 1988.

[18] Y. Chen and Z. Liu. From durational specifications to tla designs of timed automata. In *Proceedings of ICFEM04, Lecture Notes in Computer Science*, Seatle, USA, November 2004. Springer.

[19] Y. Chen and Z. Liu. Integrating temporal logics. In Eerke A. Boiten, John Derrick, and Graeme Smith, editors, *Proceedings of IFM 2004, Lecture Notes in Computer Science 2999*, pages 402–420, Canterbury, Kent, UK, 2004. Springer.

[20] E. Clarke, E.A. Emerson, and A.P. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Transactions on Programming Languages and Systems*, 8(2):244–263, 1986.

[21] E.M. Clarke, I.A. Draghicescu, and R.P. Kurshan. A unified approach for showing language containment and equivalence between various types of ω-automata. In A. Arnold and N.D. Jones, editors, *Proceedings of the 15th Colloquium on Trees in Algebra and Programming, Lecture Notes in Computer Science 431*. Springer-Verlag, Copenhagen, Denmark, 1990.

[22] J. Coenen and J. Hooman. Parameterized semantics for fault-tolerant real-time systems. In J. Vytopil, editor, *Formal Techniques in Real-Time and Fault Tolerant Systems*, pages 51–78. Kluwer Academic Publishers, Boston, 1993.

[23] F. Cristian. A rigorous approach to fault-tolerant programming. *IEEE Transactions on Software Engineering*, SE-11(1):23–31, 1985.

[24] E.W. Dijkstra. *A Discipline of Programming*. Prentice Hall, Englewood Cliff, 1976.

[25] U. Engberg, P. Grønning, and L. Lamport. Mechanical verification of concurrent systems with TLA. In G. Bochmann, editor, *Proceedings of the 4th International Conference on Computer Aided Verification, Lecture Notes in Computer Science 663*, pages 44–55. Springer-Verlag, Montreal, Canada, 1992.

[26] C.J. Fidge and A.J. Wellings. An action-based formal model for concurrent, real-time systems. *Formal Aspects of Computing*, 9(2):175–207, 1997.

[27] L. Fix and F.B. Schneider. Reason about programs by exploiting the environment. Technical Report TR94-1409, Department of Computer Science, Cornell University, Ithaca, New York 14853, 1994.

[28] An Axiomatic Basis for Computer Programming, 1969.

[29] J. He, Z. Liu, X. Li, and S. Qin. A relational model of object oriented programs. In *Proceedings of the Second ASIAN Symposium on Programming Languages and Systems (APLAS04), Lecture Notes in Computer Science*, Taiwan, March 2004. Springer.

[30] T. Henzinger, Z. Manna, and A. Pnueli. Temporal proof methodologies for timed transition systems. *Information and Computation*, 112(2):273–337, 1994.

[31] T.A. Henzinger, X. Nicollin, J. Sifakis, and S. Yovine. Symbolic model checking for real-time systems. In *Proceedings of the 7th Annual Symposium on Logic in Computer Science*, Santa Cruz, CA, USA, 1992. IEEE Computer Society.

[32] C.A.R. Hoare. *Communicating Sequential Processes*. Prentice-Hall, 1985.

[33] C.A.R. Hoare and J. He. *Unifying Theories of Programming*. Prentice-Hall, 1998.

[34] J. Hooman. *Specification and Compositional Verification of Real-Time Systems, Lecture Notes in Computer Science 558*. Springer-Verlag, Berlin, 1991.

[35] T. Janowski and M. Joseph. Dynamic scheduling in the presence of faults: specification and verification. In B. Jonsson and J. Parrow, editors, *Proceedings of 4th International Symposium on Formal Techniques in Real-Time and Fault-Tolerant Systems, Lecture Notes in Computer Science 1135*, pages 279–298. Springer, Uppsala, Sweden, 1996.

[36] T. Janowski and M. Joseph. Dynamic scheduling and fault-tolerance: Specification and verification. *Real-Time Systems*, 20(1):51 81, 2001.

[37] M. Joseph and A. Goswami. What's 'real' about real-time systems? In *Proceedings of IEEE Real-time Systems Symposium*, pages 78–85, Huntsville, Alabama, December 1988. IEEE Computer Society Press.

[38] M. Joseph and P. Pandya. Finding response times in a real-time system. *Computer Journal*, 29(5):390–395, 1986.

[39] R. Keller. Formal verification of parallel programs. *Communication of the ACM*, 19(7):371–384, 1976.

[40] R. Koymans. *Specifying message passing and Time-critical systems with temporal logic*. PhD thesis, Eindhoven University of Technology, 1989.

[41] L. Lamport. What good is temporal logic. In R.W. Mason, editor, *Proceedings of IFIP 9th World Congress*, pages 657–668, Amsterdam, the Netherlands, 1983. North-Holland.

[42] L. Lamport. Hybrid systems in TLA$^+$. In R.L. Grossman, A. Nerode, A.P. Ravn, and H. Rischel, editors, *Hybrid Systems, Lecture Notes in Computer Science 736*, pages 77–102. Springer-Verlag, 1993.

[43] L. Lamport. The temporal logic of actions. *ACM Transactions on Programming Languages and Systems*, 16(3):872–923, 1994.

[44] L. Lamport. *Specifying Systems: The TLA+ Language and Tools for Hardware and Software Engineers*. Pearson Education, Inc., 2002.

[45] L. Lamport and S. Merz. Specifying and verifying fault-tolerant systems. In H. Langmaak, W.-P. de Roever, and J. Vytopil, editors, *Proceedings of the 3rd International Symposium on Formal Techniques in Real-Time and Fault Tolerant Systems, Lecture Notes in Computer Science 863*, pages 41–76. Springer-Verlag, Lübeck, Germany, 1994.

[46] L. Lamport, R. Shostak, and Marshall Pease. The Byzantine generals problems. *ACM Transactions on Programming Languages and Systems*, 4(3):382–401, 1982.

[47] J. Lehoczky, L. Sha, and Y. Ding. The rate-monotonic scheduling algorithms: exact characterisation and average case behaviour. In *Proceedings of the 10th IEEE Real-time Systems Symposium*, pages 261–270, Santa Monica, CA, USA, 1989. IEEE Computer Society Press.

[48] X. Li, Z. Liu, J. He, and Q. Long. Generating prototypes from a UML model of requirements. In *International Conference on Distributed Computing and Internet Technology(ICDIT2004), Lecture Notes in Computer Science*, Bhubaneswar, India, 2004. Springer.

[49] C.L. Liu and J.W. Layland. Scheduling algorithms for multiprogramming in a hard real-time environment. *Journal of ACM*, 20(1):40–61, 1973.

[50] Z. Liu. *Fault-Tolerant Programming by Transformations*. PhD thesis, Department of Computer Science, University of Warwick, 1991.

[51] Z. Liu, J. He, and X. Li. rCOS: Refinement of object-oriented and component systems. In *FMCO 2004: International Symposuim on Formal Methods of Component and Object Systems. Lecture Notes in Computer SCience*, page to appear, Leiden, the Netherlands, 2004. Springer.

[52] Z. Liu, J. He, X. Li, and Y. Chen. A relational model for object-oriented requirement analysis in UML. In *Proc. of International Conference on Formal Engineering Methods, Lecture Notes in Computer Science*, Singapore, November 2003. Springer.

[53] Z. Liu and M. Joseph. Transformation of programs for fault-tolerance. *Formal Aspects of Computing*, 4(5):442–469, 1992.

[54] Z. Liu and M. Joseph. Specifying and verifying recovery in asynchronous communicating systems. In J. Vytopil, editor, *Formal Techniques in Real-Time and Fault Tolerant Systems*, pages 137–166. Kluwer Academic Publishers, Boston, 1993.

[55] Z. Liu and M. Joseph. Stepwise development of fault-tolerant reactive systems. In H. Langmaack, W.-P. de Roever, and J. Vytopil, editors, *Proceedings of the 3rd International Symposium on Formal Techniques in Real-Time and Fault Tolerant Systems, Lecture Notes in Computer Science 863*, pages 529–546. Springer-Verlag, Lübeck, Germany, September 1994.

[56] Z. Liu and M. Joseph. Verification of fault-tolerance and real-time. In *Proceedings of the 26th Annual International Symposium on Fault-Tolerant Computing*, pages 220–229, Sendai, Japan, 1996. IEEE Computer Society.

[57] Z. Liu and M. Joseph. Formalizing real-time scheduling as program refinement. In M. Bertran and T. Rus, editors, *Proceedings of Transformation-Based Reactive Systems Development, ARTS'97, Lecture Notes in Computer Science 1231*, pages 294–309. Springer, Palma, Mallorca, Spain, 1997.

[58] Z. Liu and M. Joseph. Specification and verification of fault-tolerance, timing and scheduling. *ACM Transactions on Languages and Systems*, 21(1):46–89, 1999.

[59] Z. Liu and M. Joseph. Verification, refinement and scheduling of real-time programs. *Theoretical Computer Science*, (253), 2001.

[60] Z. Liu, M. Joseph, and T. Janowski. Verification of schedulability of real-time programs. *Formal Aspects of Computing*, 7(5):510–532, 1995.

[61] Z. Liu, A.P. Ravn, and X. Li. Unifying proof methodologies of Duration Calculus and Linear Temporal Logic. *Formal Aspects of Computing*, 16(2), 2004.

[62] Q. Long, Z. Liu, J. He, and X. Li. Consistent code generation from uml models. In *Australia Conference on Software Engineering (ASWEC)*. IEEE Computer Scienty Press, 2005.

[63] G. Lowe and H. Zedan. Refinement of complex systems: A case study. Technical Report PRG-TR-2-95, Oxford University Computing Laboratory, 1995.

[64] Z. Manna and A. Pnueli. The temporal framework for concurrent programs. In R.S. Boyer and J.S. Moore, editors, *The Correctness Problem in Computer Science*, pages 215–274. Academic Press, Boston, 1981.

[65] Z. Manna and A. Pnueli. *The Temporal Logic of Reactive and Concurrent Systems: Specification*. Springer-Verlag, New York, 1991.

[66] K.L. McMillan. *Symbolic model checking – an approach to the state explosion problem*. PhD thesis, SCS, Carnegie Mellon University, 1992.

[67] R. Milner. *Communication and Concurrency*. Prentice-Hall, 1986.

[68] R. Milner. *Communication and concurrency*. Prentice Hall, 1989.

[69] A. Moitra and M. Joseph. Cooperative recovery from faults in distributed programs. In R.W. Mason, editor, *Proceedings of IFIP 9th World Congress*, pages 481–486, Amsterdam, the Netherlands, 1983. North-Holland.

[70] C. Morgan. *programming from specifications*. Prentice Hall, 1994.

[71] J. von Neumann. Probabilistic logics and the synthesis of reliable organisms from unreliable components. In C. E. Shannon and J. Macarthy, editors, *In Automata Studies*, pages 43–98, Princeton, 1956. Princeton University Press.

[72] J. Nordahl. *Specification and Design of Dependable Communicating Systems*. PhD thesis, Department of Computer Science, Technical University of Denmark, 1992.

[73] M. Pilling, A. Burns, and K. Raymond. Formal specification and proofs of inheritance protocols for real-time scheduling. *Software Engineering Journal*, 5(5), 1990.

[74] A. Pnueli. The temporal logic of programs. In *Proceedings of the 18th Annual Symposium on Foundations of Computer Science*, pages 46–57, U.S.A, 1977. IEEE Computer Society Press.

[75] A. Pnueli. Applications of temporal logic to the specification and verification of reactive systems: A survey of current trends. In J.W. de Bakker, W.-P. de Roever, and G. Rozenberg, editors, *Current Trends in Concurrency, Lecture Notes in Computer Science 224*, pages 510–584. Springer-Verlag, Berlin, Heidelberg, New York, 1986.

[76] A. Pnueli and E. Harel. Applications of temporal logic to the specification of real-time systems. In M. Joseph, editor, *Proceedings of the 1st International Symposium on Formal Techniques in Real-Time and Fault-Tolerant Systems, Lecture Notes in Computer Science 331*, pages 84–98. Springer-Verlag, Warwick, U.K., 1988.

[77] B. Randell. System structure for software fault tolerance. *IEEE Transactions on Software Engineering*, SE-1(2):220–232, June 1975.

[78] B. Randell, P.A. Lee, and P.C. Treleaven. Reliability issues in computing systems design. *Computing Survey*, 10(2):123–165, 1978.

[79] A.P. Ravn, H. Rischel, and K.M. Hansen. Specifying and verifying requirements of real-time systems. *IEEE Transactions on Software Engineering*, 19(1):41–55, 1993.

[80] H. Schepers and R. Gerth. A compositional proof theory for fault-tolerant real-time systems. In *Proceedings of the 12th Symposium on Reliable Distributed Systems*, pages 34–43, Princeton, NJ, 1993. IEEE Computer Society Press.

[81] R.D. Schlichting and F.B. Schneider. Fail-stop processors: an approach to designing fault tolerant computing systems. *ACM Transactions on Computer Systems*, 1(3):222–238, 1983.

[82] U. Shrotri, P. Bhaduri, and R. Venkatesh. Model checking visual specification of requirements. In *International Conference on Software Engineering and Formal Methods (SEFM 2003)*, page 202209, Brisbane, Australia. IEEE Computer Society Press.

[83] J. Yang, Q. Long, Z. Liu, and X. Li. A predicative semantic model for integrating UML models. In *Proc. Theoretical Aspects of Computing - ICTAC 2004, First International Colloquium, 21-24 September 2004, Guiyang, China, Revised Selected Papers. Lecture Notes in Computer Science 3407*. Springer, 2005.

[84] Y. Zhang and C.C. Zhou. A formal proof of the deadline driven scheduler. In H. Langmaak, W.-P. de Roever, and J. Vytopil, editors, *Proceedings of the 3rd International Symposium on Formal Techniques in Real-Time and Fault Tolerant Systems, Lecture Notes in Computer Science 863*, pages 756–775. Springer-Verlag, Lübeck, Germany, 1994.

[85] C.C. Zhou, M.R. Hansen, A.P. Ravn, and H. Rischel. Duration specifications for shared processors. In J. Vytopil, editor, *Proceedings of the 2nd International Symposium on Formal Techniques in Real-Time and Fault Tolerant Systems, Lecture Notes in Computer Science 571*. Springer-Verlag, Nijmegen, the Netherlands, January 1992.

[86] C.C. Zhou, C.A.R. Hoare, and A.P. Ravn. A calculus of durations. *Information Processing Letters*, 40(5):269–276, December 1991.