Real-Time and Fault-Tolerant Systems
–Specification, Verification, Refinement and Schedulability

Zhiming Liu

The United Nations University

International Institute for Software Technology

P. O. Box 3058, Macau

http://www.iist.unu.edu/~lzm

Joint work with

Prof. Mathai Joseph, Tata Research Development and Design Centre, India.

Prof. Anders Ravn, Aalborg University, Denmark

Dr. Xiaoshan Li, University of Macao

Outline

1. Introduction
2. Real-Time systems: An Informal Account
3. Formal Specification, Verification and Refinement
4. Fault-Tolerance as Refinement
5. Formal Treatment of Real-Time
6. Both Fault-Tolerance and Time
7. Schedulability and Fault-Tolerant Schedulability as Refinement
8. Conclusions

INTRODUCTION

About the development, understanding and use of formal techniques for the specification, verification, and construction of *safety critical computer systems*:

- concurrent and distributed systems, real-time systems, communication networks, air-traffic control systems, etc.

The correct and reliable construction of these systems is concerned with the inter-related issues of

- *concurrency*, *fault-tolerance*, *timing*, and *schedulability analysis*

each of which can and does indeed form its own research field.

Historical Background

Since the 1970's, formal techniques for the design and analysis of concurrent systems have seen considerable development. This have made a significant contribution to

- a better understanding of the behaviour of concurrent systems, and thus
- more reliable construction of concurrent systems.

Most Widely Studied Methods

- *Transition systems* + *Temporal logic* [Pnueli 77; Manna and Pnueli 81, Lamport 83, Lamport 94]
- *Automata* + *Temporal logic* [Clarke et al. 86, 90]
- *Process algebras* [Hoare's CSP 78; Milner's CCS 80]

Question: Should and can we use these methods and their tools consistently in one development project? – UTP, rCOS $\rightsquigarrow$ Theory of CBSD

Common Features of the Traditional Frameworks

1. Used as an approach for *fault avoidance* and *fault-removal* – *fault-intolerance*
   - a programming language has a fixed semantics,
   - the *correctness* of a program is conditional – executing on a *fault-free* system.
2. Use a discrete event approach, abstract away the time and describe only the ordering of the events.

   Typically concerned with *qualitative temporal properties*:
   - *safety*: $\varphi \Rightarrow \Box\psi$
   - *liveness*: $\varphi \Rightarrow \Diamond\psi$

Fault-Tolerance

- Another approach towards to the construction of safe and dependable systems does not assume a fault-free system
- It incorporates additional components to ensure that the occurrence of an *error state* does **NOT** result in later *failures* [Neumann 56, Randell 75, Avizienis 76]

NB

1. Little work on formal treatment of fault-tolerance until the 1980's [Schilichting and Schneider 83; Cristian 85].
2. Critical system design requires *how a system is reliable enough* to be precisely specified and verified, i.e. a **formal method for verifying fault-tolerance** to

   formally model the effect of faults and verify that the system meets its requirement in the presence of the faults.

Real-Time

A *real-time system* does not only require that events must occur in a required order, but also occur timingly

- an event must not occur too late
  - *bounded response*: $\varphi \stackrel{\varepsilon}{\leadsto} \psi$
- an event must not occur too soon
  - *bounded invariance*: $\varphi \Rightarrow \Box_\varepsilon \psi$

NB:

1. Formal development of real-time systems requires the *quantitative temporal properties* to be specified and reasoned about.
2. Since the late 1980's, all the widely used traditional formal frameworks have been extended to deal with timing.

Schedulability

The issue of schedulability arises when a real-time program is to be implemented on a system with *limited resources*, such as processors.

The Problem: Given

(a). a description of program P of n processes with timing constraints,

(b). a description of a scheduler (its policy) S, and

(c). a set Pr of m processors with their speed,

how can it be proved that P satisfies its timing constraints when the processors are shared between the processes by the scheduler S?

NB: *An infeasible implementation of a real-time program will not meet it timing requirement even though the program has been proven correct.*

Existing Solutions

- Scheduling analysis theory
  1. It is limited by all kinds of hypotheses, e.g. all tasks are periodic, all tasks are independent from each other.
  2. Models are different from models for program development
- Formal specification [Henzinger et al. 91; Hooman 91; etc.]
  1. Scheduling and program development are mixed.
  2. When dealing with pre-emption, a scheduled operation is to be *explicitly* divided into smaller operations

Results from scheduling analysis theory cannot be directly used in the system development model

Problems to Solve

- How to bridge the gap between the real-time program development and scheduling analysis in a complete real-time system development?
- How to reason about and make formal use of the methods and results from the scheduling theory?
- How to construct a feasible scheduler for a given program, if there is one?

Overview

Study how fault-tolerance, real-time and schedulability, as well as functional correctness, can be specified and verified within a single formal framework

The Approach

- Use of an existing formalism (TLA) to model fault-tolerance, timing, and scheduling
- Use of existing techniques of verification, refinement and transformation for formal design of fault-tolerant real-time systems, as well as for formal schedulability analysis.
- Show how the methods and results in the scheduling analysis are formally proved and used in a formal development framework.

REAL-TIME SYSTEMS — An Informal Account

Consider a system in which

- a computer controls a device or a process through actuators
- a sensor provides readings (e.g at periodic intervals)
- the computer must respond by sending signals to actuators
- the computer may also have to respond to unexpected or irregular events

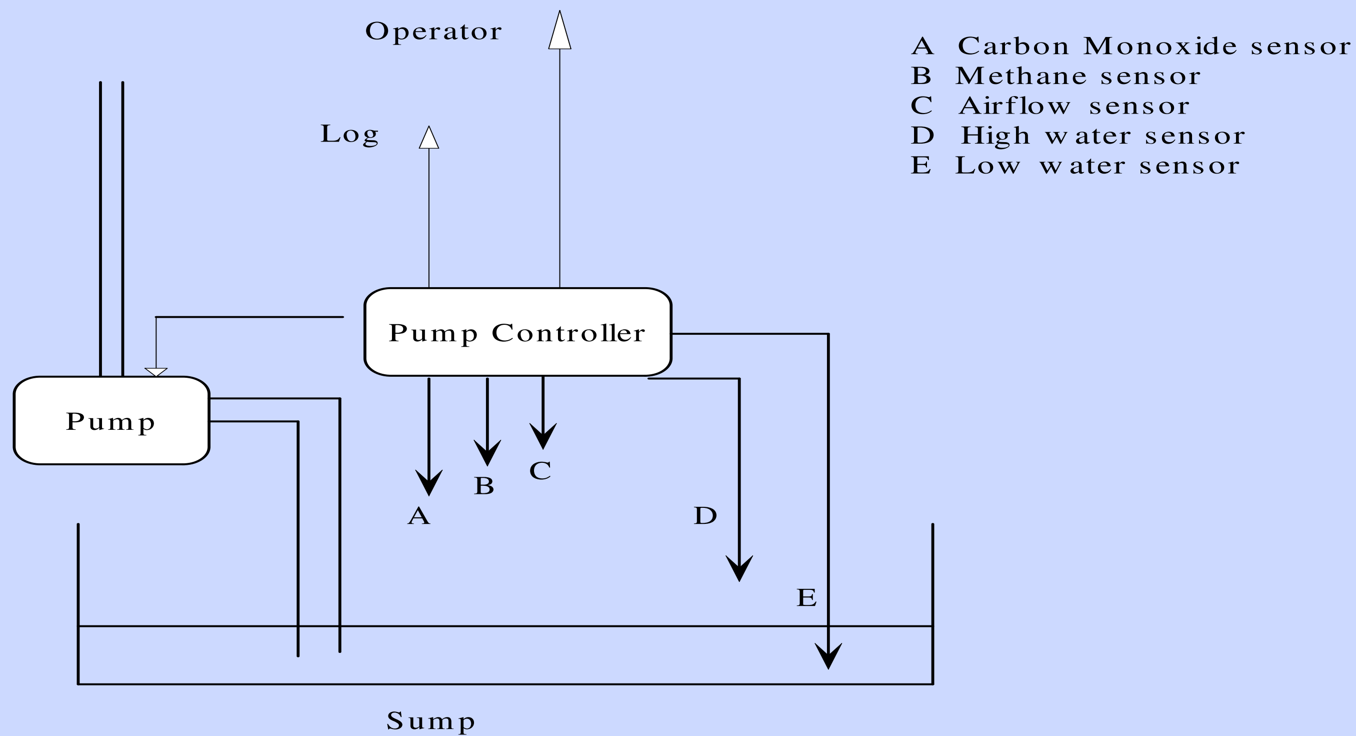
Limited Resources

- A responses must be delivered with a time-bound
- If a number of events occur close together, the computer needs to schedule the computations
- If this is not possible to achieve, we say that the system lacks sufficient resource.
- A system with unlimited resource and capable of processing at infinite speed could satisfy any time constraints

Consequences of Missing Deadlines

- No effect at all no deadlines needed – Untimed Systems
- The effects are minor and correctable – soft deadlines
- The results are catastrophic – hard deadlines

A Mine Pump System



Real-Time Programs

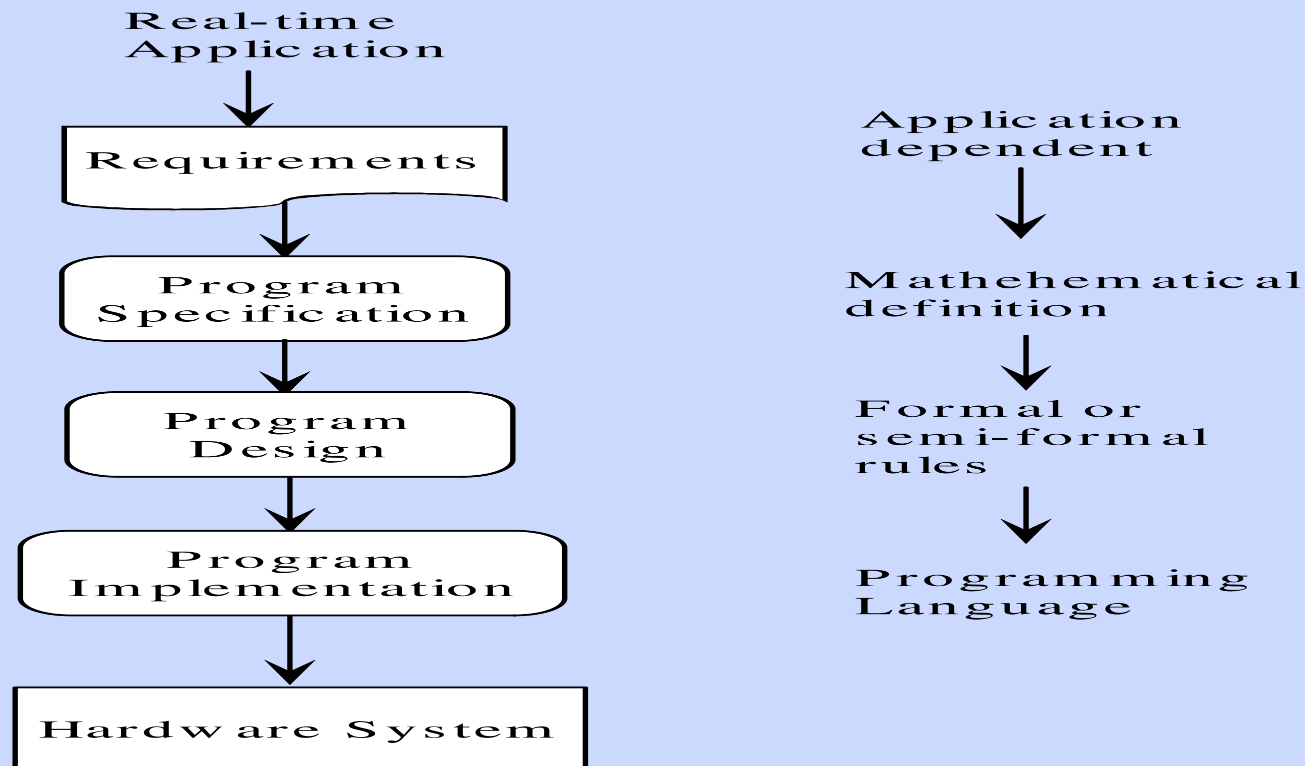
A real-time program must

- interacts with an environment which has time varying properties, e.g. car, water level
- exhibits predicable time-dependent behavior
- executes on a system with limited resources

Hard to Predict Timing

- A task may take different time under different conditions
- Tasks may have dependencies
- More than one processors may be needed in a system
- The nature of the application may require distributed computing, with nodes connected by communication lines
- Communication takes time

System Development



Requirements

- The demands placed on a real-time system is called the requirements.
- Real-time requirements are application dependent
- Requirements include functional requirements and nonfunctional requirements, such as timing properties
- The functional and non-functional requirements must be precisely defined and together used to construct the specification of the system

Specification

- A specification is a statement of the properties to be exhibited by the system
- It states what the system should do within what timing bounds
- It does **not** say how the system does them
- Mathematical description and analysis are important in dealing with the complexity of real problems

Is Mathematics too difficult? What can we do without it?

The Mine Pump System

- 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 airflow level is critical
- Operation requirement
  1. The mine is operated for three shifts a day
  2. For not more than one shift in 1000 to be lost due to high water level

Problem

Consider the following specifications:

1. The mine must always not violate the safety requirements
2. The mine must not be operated when the safety requirements are violated
3. The mine must always satisfy both the safety and operation requirements

Which one do we want?

The Goal of Requirement Analysis

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 and operational requirements

Safety Properties and Progress Properties

Assumptions

- There are often assumptions without which the requirements cannot be met:
  1. what if the methane level can rise arbitrarily fast?
  2. what if the rate of change of the water level is unbounded?
- Note that
  1. the sensors operate by sampling at periodic interval
  2. the computer will take time to perform computation to send commands
  3. the pump will take time to start and stop
- The specification should also clearly state these assumptions

What vs How

- Under what conditions the mine must be evacuated, the pump must start or stop
- But not how and when these and how often these should be done, i.e. no information about
  1. how often the mine must be evacuated
  2. how normal operation is resumed after an evacuation
- These are design or implementation decisions to be made to meet the requirements

Develop a specification

1. Describe the requirements as properties, using math notation
2. The often used math notions and symbols include:
   (a) Predicates, logical operators $\vee$, $\wedge$, $\Rightarrow$, $\neg$, $\forall$, $\exists$, etc.
   (b) Mathematical relations and functions, constants, variables and intervals

$$F : T1 \longrightarrow T2, \;\; V : T, \;\; [b, e], \;\; [b, e), \;\; (b, e], \;\; (b, e)$$

Variables of the Mine Pump System

Water

- Let $Water$ represent the water level at any time

$$Water : Time \longrightarrow Real$$

- Let $WaterIn$ and $WaterOut$ represent the rates at which water enters and leaves the sump

$$WaterIn, WaterOut : \ Time \longrightarrow Real$$

- The depth of water at time $t_2 > t_1$, $Water(t_2)$ is

$$\begin{aligned} Water(t_2) = \quad & Water(t_1) + \\ & \int_{t_1}^{t_2} (WaterIn(t) - WaterOut(t))dt \end{aligned}$$

Water Continued

- $HighWater$ and $LowWater$ represent the high and low sensor positions

$$LowWater < HighWater < DangerWater$$

- If $HighWater = LowWater$, then only one sensor is needed

Methane Level

- The level of methane

$$Methane : Time \longrightarrow Real$$

- $DangerMethane$ represents the critical methane level
- Rates at which the methane flows in and out

$$MethaneIn, MethaneOut : Time \longrightarrow Real$$

- For all t_1, t_2,

$$\begin{aligned} Methane(t_2) = \quad & Methane(t_1) + \\ & \int_{t_1}^{t_2} (MethaneIn(t) - MethaneOut(t))dt \end{aligned}$$

Formal Description of the Assumptions

1. $WaterIn(t) \leq MaxWaterIn.$
2. $MaxWaterIn < PumpRating.$
3. $(Pumping(t) \wedge Water(t) > 0) \Rightarrow (WaterOut(t) > PumpRating)$
4. Enough time to react before water becomes danger

   $(HighWater + MaxWaterIn \cdot (t_P)) < DangerWater$
5. Enough time to react before methane becomes dangerous

   $(HighMethane + MaxMethaneRate \cdot (t_P)) < DangerMethane$
6. The methane level does not reach $HighMethane$ more than once in 1000 shifts; without this limit, it is not possible to meet the operational requirement.

The Pump Controller

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

Refinement: Reaction Time

Assume t_P is the reaction time of the pump:

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

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

Sensors

$$\forall t \in Time \cdot \quad Water(t) \geq HighWater \Rightarrow HW(t)$$
$$\wedge Water(t) \geq LowWater \Rightarrow LW(t)$$

$$\forall t \in Time \cdot \quad Methane(t) \geq DangerMethane \Rightarrow HM(t)$$
$$Methane(t) < DangerMethane \Rightarrow LM(t)$$

Actuator

$$PumpOn(t) \Rightarrow \exists t_o \leq t_P \cdot Pumping(t + t_0)$$

$$PumpOff(t) \Rightarrow \exists t_o \leq t_P \cdot \neg Pumping(t + t_0)$$

$PumpOn$ is set by the controller

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.

A Refined Controller and Actuator

- Controller

  $HW(t) \wedge LM(t) \Rightarrow \exists t_o \leq \varepsilon \cdot PumpOn(t + t_o)$

  $HW(t) \wedge HM(t) \Rightarrow \exists t_o \leq \varepsilon \cdot PumpOff(t + t_o)$

- Pump

  $PumpOn(t) \Rightarrow \exists t_o \leq \kappa \cdot Pumping(t + t_0)$

  $PumpOff(t) \Rightarrow \exists t_o \leq \kappa \cdot \neg Pumping(t + t_0)$

- Assume $\varepsilon + \kappa \leq t_P$

Verification

Prove

$$Assumption \wedge (\text{Control Specification}) \wedge Actuator \wedge Sensors$$

$$\Downarrow$$

$$\text{ReqSpec}$$

FORMAL SPECIFICATION, VERIFICATION AND REFINEMENT

- TLA
- The computational model
- Programs and their specification.
- Verification and refinement.
- Program development by refinement

Introducing TLA

- TLA is a logic used for specifying and reasoning about programs manipulating data
  - Assume an infinite set of values *Val*
- A program manipulates data by changing its states
- A state assigns values to state variables
  - Assume an infinite set of *Var* of state variables, represented by x, y, z, etc.
  - For $\overline{v} \subseteq \mathit{Var}$, a state over $\overline{v}$ is a mapping from $\overline{v}$ to *Val*
  - For a state s, $s[x]$ and $s[\overline{v}]$ are the values of x and $\overline{v}$ in s

Examples of States

- State variables: $\{x, y\}$
  - $s = \{x \mapsto 0, y \mapsto 1\}$, $s' = \{x \mapsto 1, y \mapsto 0\}$
  - $s[x] = 0$, $s[y] = 1$, $s'[x] = 1$, $s'[y] = 0$
- $\{On, Off, Bright\}$ – state variables of a light
  - $s_1 = \{Off \mapsto true, On \mapsto false, Bright \mapsto false\}$
    $s_2 = \{Off \mapsto false, On \mapsto true, Bright \mapsto false\}$
    $s_3 = \{Off \mapsto false, On \mapsto ture, Bright \mapsto true\}$
  - $s_1[On] =$?......
- Let st be a variable and $\{on, off, bright\}$ are values
  $s_1 = \{st \mapsto off\},\ s_2 = \{st \mapsto on\},\ s_3 = \{st \mapsto bright\}$

State Predicates

- Properties of a state are described by first order predicate formulas
- A state predicate is a formula built from state variables and values in the first order predicate calculus.
- For examples, $x - 1 = y$ and $x + y > 3$ are predicates.
- Given an interpretation to the predicate symbols and the function symbols, the meaning of a predicate Q is a mapping $[\![Q]\!]$ from the states to the Booleans $\{\mathit{true}, \mathit{false}\}$.
- A state s satisfies a predicate Q, denoted by

$$s \models Q \;\; \mathit{iff} \;\; [\![Q]\!](s) = \mathit{true}$$

Examples of Predicates

- $\{x, y\}$ are the variables
- $s = \{x \mapsto 0, y \mapsto 1\}$, $s' = \{x \mapsto 1, y \mapsto 0\}$
- $P : x - 1 = y$,

  $Q : x + y > 3$,

  $R : (x - 1 = y) \vee (x + y > 3)$
- $s \models ?$..........

Actions

A program changes states from one to another by carrying out atomic actions

- An action is a first-order predicate over *Var* and their 'primed versions' *Var*′
- For examples, $x' + 1 = y$ and $x \geq y' + (x - 1)$.
- Both $x' + 1 = y$ and $x' = y - 1$ model $x := y - 1$
- $x \geq y' + (x - 1)$ represents the relation between the values of x and y after the execution and the value of x before the execution of the action.

Semantics of Actions

- For a given interpretation, the meaning of an action τ assigns a Boolean value to a pair (s, s') of states:

$$[\![\tau]\!](s, s') = \textit{true} \;\; \textit{iff} \;\; (s[z]/z, s'[z]/z')$$

- A pair (s, s') satisfies an action, denoted by

$$(s, s') \models \tau, \;\; \textit{iff} \;\; [\![\tau]\!](s, s') = \textit{true}$$

Enabling Condition

- A predicate Q can also be viewed as a particular action which does not have primed variables:

$$(s, s') \models Q \text{ iff } s \models Q$$

- For an action τ, let $x'_1, \ldots, x'_n$ be the primed variables that occur in τ, let $\hat{x}_1, \ldots, \hat{x}_n$ be new *rigid* variables that do not occur in τ,

$$en(\tau) \stackrel{def}{=} \exists \hat{x}_1, \ldots \hat{x}_n . \tau[\hat{x}_1/x'_1 \ldots \hat{x}_n/x'_n]$$

- $en(\tau)$ is a predicate, called the *enabling condition* of τ,

$en((x > 0) \wedge (x' = x - 1)) =?$

$en((x > 0) \wedge (x' = x - 1) \vee (x < 0) \wedge (x' = x + 1)) =?$

Actions in Light Control

- Turn on the light: $Off \wedge On'$
- Turn off the light: $On \wedge Off'$
- Brighten the light: $On \wedge \neg Bright \wedge Bright'$

1. No specification about how to turn off from state On and $Bright$
2. The light can be brightened when it is On

$(s_1, s_2) \models$ Turn on the light ....

What are the actions for the version with the variable st?

Temporal Formulas

- Built from actions as the elementary temporal formulas using Boolean connectives and modal operators in Linear-Time Temporal Logic.
- Boolean connectives: $\vee$, $\wedge$, $\neg$, $\Rightarrow$
- We only use the temporal operator $\Box$ and its derivables (e.g. $\Diamond$).
- Quantifications (i.e. $\exists$, $\forall$) are possible over a set of both rigid variables and state variables.

Semantics

- Formulas are used to describe and reason about infinite state sequences
- For an infinite sequence $\sigma = \sigma_0, \sigma_1, \ldots\ldots$

  $$[\![\tau]\!](\sigma) = \mathit{true} \;\; \mathit{iff} \;\; [\![\tau]\!](\sigma_0, \sigma_1) = \mathit{true}$$

- The first-order connectives and quantification over rigid variables retain their standard semantics
- $[\![\Box\varphi)]\!](\sigma) = \mathit{true} \;\; \mathit{iff} \;\; [\![\varphi]\!](\eta) = \mathit{true}$ for any suffix η of σ
- $[\![\exists x.\varphi]\!](\sigma) = \mathit{true}$ *iff* there is η such that $\eta =_x \sigma$ and $[\![\varphi]\!](\eta) = \mathit{true}$

Examples

- $\{x, y\}$ are the variables
  - $s_1 = \{x \mapsto 0, y \mapsto 1\}$, $s_2 = \{x \mapsto 1, y \mapsto 0\}$
- $\sigma = s_1, s_2, s_1, s_2.............$
- $\sigma \models (x = y - 1)$
- $\sigma \models \Box(x + 1 = 1)$
- $\sigma \models \Box(y + y' = 1)$
- $\sigma \models \exists x.\Box((y \geq 0) \wedge (x = 1))$
- $\sigma \models \Box\Diamond(y' = y - 1)$,

  $\sigma \models \Box\Diamond(y' = y + 1)$,
- $\sigma \models \Box(\Diamond(y' = y - 1) \wedge \Diamond(y' = y + 1))$?

Proof System

- A formula φ is satisfied by σ, denoted by

$$\sigma \models \varphi, \;\; \mathit{if} \;\; [\![\varphi]\!](\sigma) = \mathit{true}$$

- φ is valid if it is satisfied by any infinite state sequences over *Var*.
- A sound and relatively complete proof system is given in [Lamport 94]
- Every valid formula is provable from the axioms and proof rules of the proof system if all the valid action formulas are provable.

Some Valid Formulas

- The following are valid
  - $\Box(\varphi \vee \neg\varphi)$
  - $\Box\varphi \Rightarrow \varphi$, $\varphi \Rightarrow \Diamond\varphi$, $\Box\varphi \Rightarrow \Diamond\varphi$
  - $\Diamond(\varphi \vee \psi) \Rightarrow \Diamond\varphi \vee \Diamond\psi$
- **How about the other way around**?
  - $\Diamond(\varphi \wedge \psi) \Rightarrow \Diamond\varphi \wedge \Diamond\psi$
  - $\Box(\varphi \vee \psi) \Rightarrow \Box\varphi \vee \Box\psi$
  - $\Box(\varphi \wedge \psi) \Rightarrow \Box\varphi \wedge \Box\psi$

Computational Model: Action Systems

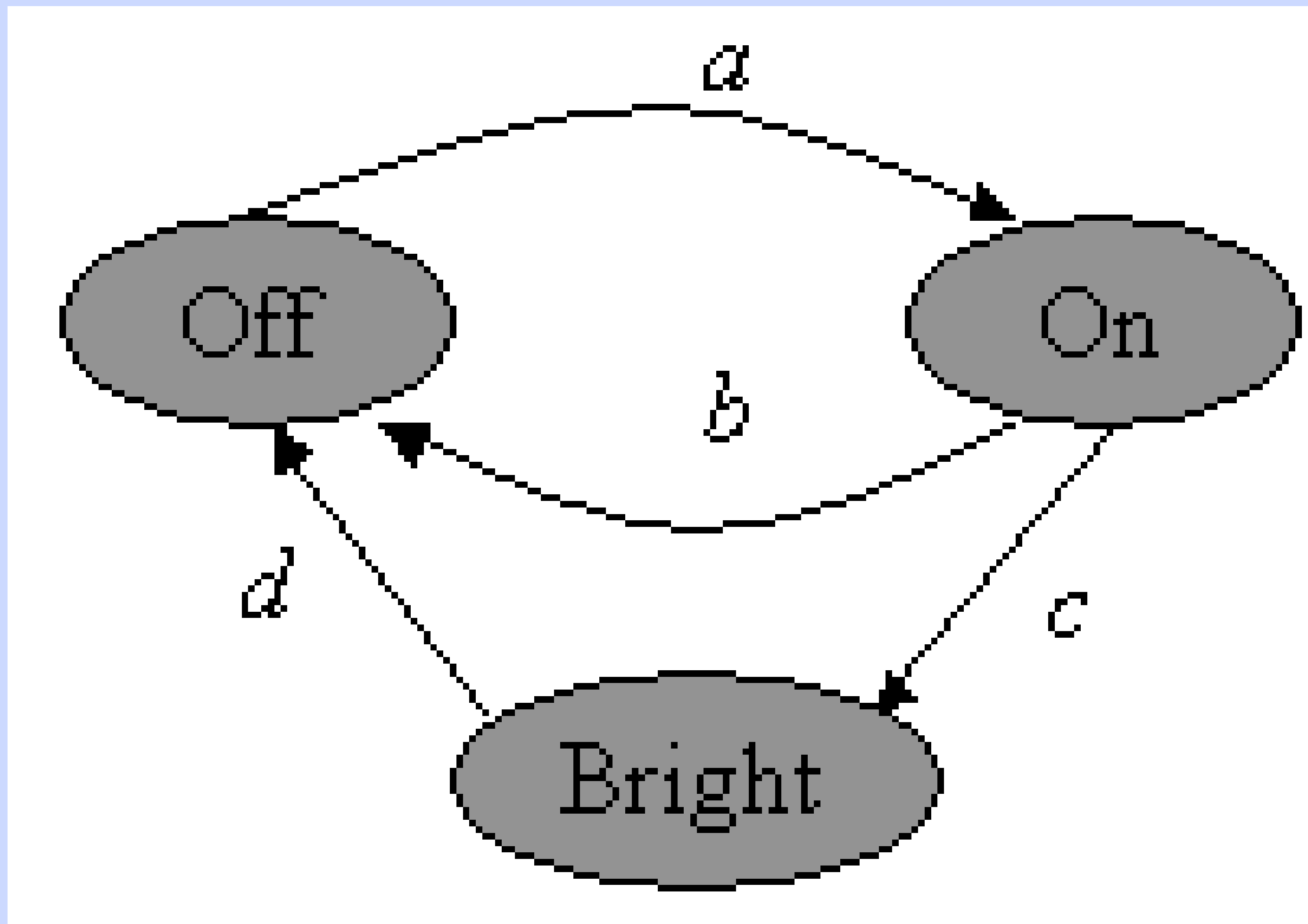
- A program is represented as an action system
- An action system is a tuple $P = (\overline{v}, \overline{x}, \theta, A)$ consisting of four components, where
  - $\overline{v}$ is a finite set of state variables
  - $\overline{x} \subseteq \overline{v}$ is a set of internal variables
  - θ is a predicate over $\overline{v}$ called the initial condition
  - A is a finite set of atomic actions over $\overline{v}$

A Light Control System

- $\overline{v} \stackrel{def}{=} \{st\}, \overline{x} \stackrel{def}{=} \{\ \}$
- $\Theta \stackrel{def}{=} st = off$
- $A \stackrel{def}{=} \left\{ \begin{array}{l} a : (st = off) \wedge (st' = on), \\ b : (st = on) \wedge (st' = off), \\ c : (st = on) \wedge (st' = bright), \\ d : (st = bright) \wedge (st' = off) \end{array} \right\}$

At anytime, st can only take values in $\{on, off, bright\}$, but can this be formally specified and proved?

Graphic Representation –Sate Machines



Open Systems and Composition

- Light control system: $LightC$

  $(st = off \wedge button = pressed) \wedge (st' = on \wedge button' = released)$

  $(st = on \wedge button = pressed) \wedge (st' = off \wedge button' = released)$

  $(st = on \wedge button = pressed) \wedge (st' = bright \wedge button' = released)$

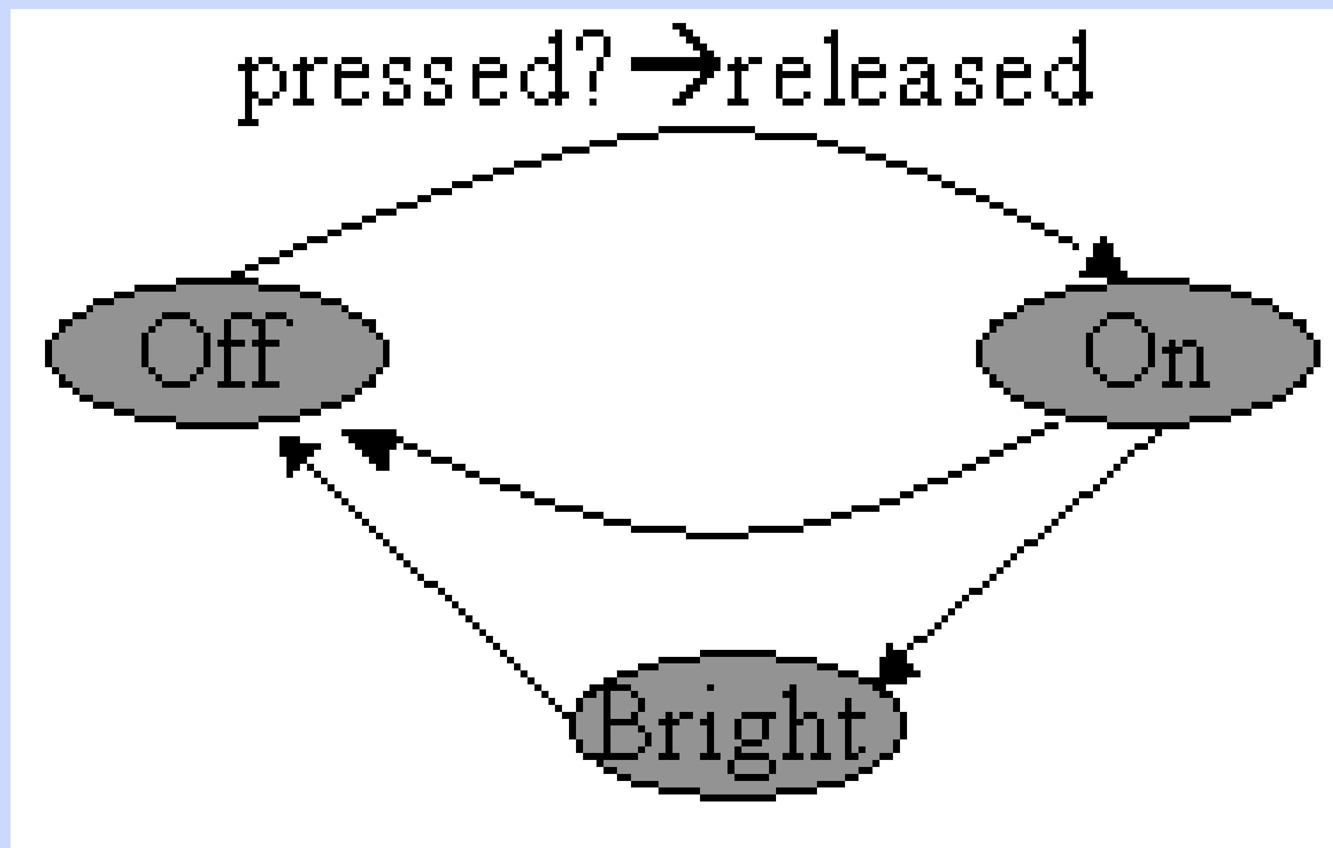
  $(st = bright \wedge button = pressed) \wedge (st' = off \wedge button' = released)$

- The Button: $Button$

  $(button = released) \wedge (button' = pressed)$

What is $LightC \parallel Button$?

Composing machines



Computation

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

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.

An action system satisfies a property φ if all its computations satisfies φ.

How to prove the light control systems satisfies $\Box(st \in \{on, off, bright\})$?

Remarks on the model

- A computation either contains infinitely many diligent steps, or a diligent step takes it to a terminating state after which only stuttering steps occur.
- The set of all the computations of a program is *stuttering closed*.

Why Stuttering?

Why Stuttering?

Consider a digital clock that displays only the hour. Let hr represents the clock's display.

The 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

$$\mathit{HCnext} \stackrel{def}{=} hr' = (hr \bmod 12) + 1$$

The clock can be specified by $\mathit{HCinit} \wedge \Box \mathit{HCnext}$

This would be fine if the clock is considered in isolation and never be related to another system.

How can we model a device that displays the current hour and temperature?

Stuttering is Essential for Composition and Refinement

$$\left\{ \begin{array}{l} hr \mapsto 11, \\ temp \mapsto 23.5 \end{array} \right\} \longrightarrow \left\{ \begin{array}{l} hr \mapsto 12, \\ temp \mapsto 23.5 \end{array} \right\} \longrightarrow$$
$$\left\{ \begin{array}{l} hr \mapsto 12, \\ temp \mapsto 23 \end{array} \right\} \longrightarrow \left\{ \begin{array}{l} hr \mapsto 12, \\ temp \mapsto 22.5 \end{array} \right\} \ldots$$

ALSO, stuttering closure property is essential for **refinement**

Program Specification – Basics

- The full specification of an action τ always contains a conjunct

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

- E.G., the guarded command $x > 0 \rightarrow x := x - 1$ is defined as

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

- Omit the *unchanged* part when when no confusion
- To specify stuttering, for an action τ and a finite set of variables $\overline{z}$

$$[\tau]_{\overline{z}} \stackrel{def}{=} \tau \vee unchanged(\overline{z})$$

Program Specification

Given a program $P = (\overline{v}, \overline{x}, \Theta, A)$

- The *state-transition relation*: $\mathcal{N}_P \stackrel{def}{=} \bigvee_{\tau \in A} \tau$
- The *exact specification*: $\Pi(P) \stackrel{def}{=} \Theta \wedge \Box[\mathcal{N}_P]_{\overline{v}}$
- The *canonical safety specification*: $\Phi(P) \stackrel{def}{=} \exists \overline{x}.\Pi(P)$
- $\Pi(HC) = \mathit{HCinit} \wedge \Box[\mathit{HCnext}]_{hr}$ rules out behaviours like

$$\{hr \mapsto 11\} \longrightarrow \{hr \mapsto 5\} \longrightarrow \{hr \mapsto 6.3\} \longrightarrow \{hr \mapsto 12\} \cdots$$

Healthiness Condition: $\Theta \wedge \Box[\mathcal{N}_P]_{\overline{v}} \equiv \Theta \wedge \Box[\mathcal{N}_P \vee Skip]_{\overline{v}}$

Fairness Conditions

$\Pi(P)$ and $\Phi(P)$ are usually strengthened by conjoining them with one or more fairness properties of the forms

Weak fairness: $WF_{\overline{z}}(\tau) \stackrel{def}{=} (\Box\Diamond\langle\tau\rangle_{\overline{z}}) \vee (\Box\Diamond\neg en(\langle\tau\rangle_{\overline{z}}))$

Strong fairness: $SF_{\overline{z}}(\tau) \stackrel{def}{=} (\Box\Diamond\langle\tau\rangle_{\overline{z}}) \vee (\Diamond\Box\neg en(\langle\tau\rangle_{\overline{z}}))$

where $\langle\tau\rangle_{\overline{z}} \stackrel{def}{=} \tau \wedge \neg unchanged(\overline{z})$

Adding the fairness condition of *HCnext* ensures that the clock will eventually move forward

Linking Theories of Programming to TLA

- A TLA action can be implemented by a guarded command $g \longrightarrow C$

  $$C ::= x := e \mid C; C \mid C \sqcap C \mid C \lhd b \lhd \rhd C \mid b * C$$

- If a command is given, we can calculate the design $[\![C]\!]$ of C

  $$Pre_C \vdash Post_C$$

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

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

- We can use $(Pre_C \Rightarrow Post_C) \lhd g \rhd Skip$, as we allow stuttering
- If the UTP design of $g \longrightarrow C$ is $\neg okay$, use $false$ or $skip$ as the corresponding TLA action

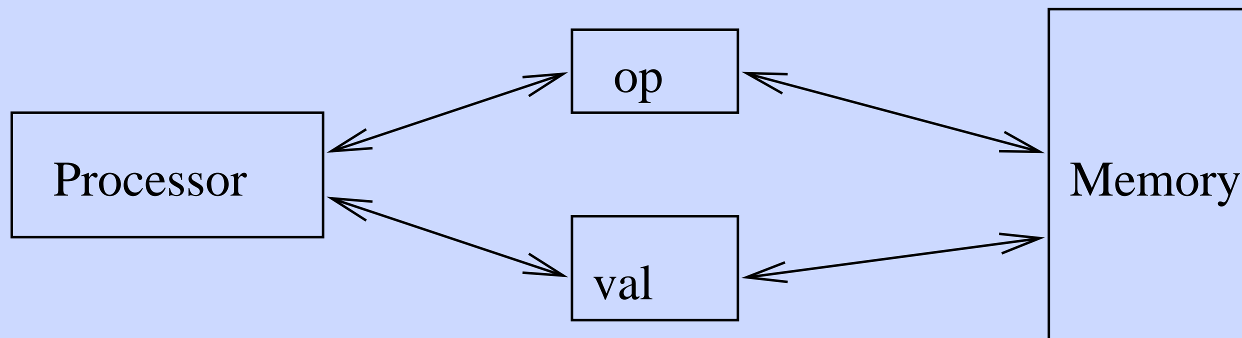
But, an atomic action does NOT have to be implemented by a sequential command

Exercises

1. Define the Morgan's specification statement $w : [p; q]$, assuming x and x' represent the initial value and final value of x.
2. Define the non-deterministic choice $\tau_1 \sqcap \tau_2$
3. Define $\{p\}\tau\{q\}$ as a TLA action
4. Understand how does the notation unify the semantics of deterministic choice and non-deterministic choice.

Example

A processor-memory interface



- *op* is set by the processor and cleared by the memory after the operation is over; *op* take values in $\{rdy, r, w\}$.
- val is set by the processor for a write and by the memory for a read, assuming its value space $\mathbf{Z}$.

Specification of the Interface Program

$$
\begin{array}{lcl}
\overline{v}_1 & \stackrel{def}{=} & \{op, val, d\}, \overline{x}_1 \stackrel{def}{=} \{d\} \\
\Theta_1 & \stackrel{def}{=} & (op = w) \wedge val \in \mathbf{Z} \\
R_1^p & \stackrel{def}{=} & (op = rdy) \wedge (op' = r) \\
W_1^p & \stackrel{def}{=} & (op = rdy) \wedge (op' = w) \wedge (val' \in \mathbf{Z}) \\
R_1^m & \stackrel{def}{=} & (op = r) \wedge (op' = rdy) \wedge (val' = d) \\
W_1^m & \stackrel{def}{=} & (op = w) \wedge (op' = rdy) \wedge (d' = val) \\
A_1 & = & \{R_1^p, W_1^p, R_1^m, W_1^m\} \\
P_1 & = & (\overline{v}_1, \Theta_1, A_1) \\
\mathcal{N}_{P_1} & = & R_1^p \vee W_1^p \vee R_1^m \vee W_1^m \\
\Pi(P_1) & = & \Theta_1 \wedge \Box[\mathcal{N}_{P_1}]_{\overline{v}_1} \\
\Phi(P_1) & = & \exists d.\Theta_1 \wedge \Box[\mathcal{N}_{P_1}]_{\overline{v}_1}
\end{array}
$$

Remark

R_1^p and W_1^p can be combined into a single nondeterministic action:

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

Verification

- To prove P has a property ψ is to prove the implication

  $$\Phi(P) \Rightarrow \psi$$

- To prove that P satisfies $\Box p$ is to prove that $\Theta \Rightarrow p$, and for each $\tau \in P$,

  $$\{p\}\tau\{p\}$$

  p is called an **invariant** of P.

- E.G: $P = (\{x\}, x = 2, \{x' = x + 2, x' = 2 * x\})$ satisfies

  $$\Box(x > 0) \wedge \Box(x \text{ is even})$$

  Can you try to prove it?

The Interface Program Example Continued

We can prove that P_1 satisfies

$$\exists d.\Box((op \neq w) \Rightarrow (d = val))$$

$$\Box(val \in \mathbf{Z})$$

TLA is a untyped logic, but *correct typing* is specified and reasoned about as an invariant.

REMARKS

- For a concurrent program with an action τ implemented as a guarded command $g_i \longrightarrow C_i$, $\{p\}\tau\{q\}$ can be proven by proving in UTP

  $$\{g \wedge p\}\tau\{q\}$$

  Or directly by Hoare Logic, or Wp calculus

- Abstract sequential programming away, but allows any langauge and theories for sequential programming to be embedded!

- Reason about temporal properties by non-temporal reasoning

Refinement

Given two programs $P_l = (\overline{v}_l, \overline{x}, \Theta_l, A_l)$ and $P_h = (\overline{v}_h, \overline{y}, \Theta_h, A_h)$

The *refinement relation* $P_l \sqsupseteq P_h$ holds if $\Phi(P_l) \Rightarrow \Phi(P_h)$

Proving Refinement by simulation:

1. define state functions $\tilde{y}_i$ for all $y_i \in \overline{y}$ in terms of the variables $\overline{v}_l$, and
2. 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 y_i in $\Pi(P_h)$, by proving:
   (a) *initiality-preservation:* $\Theta_l \Rightarrow \widetilde{\Theta}_h$;
   (b) *step-simulation:* $\mathcal{N}_{P_l} \Rightarrow [\widetilde{\mathcal{N}}_{P_h}]_{\widetilde{\overline{v}}_l}$.

Allows any theories refinement for sequential programming to be embedded into the theory.

Refinement with Liveness Properties

- A refinement which preserves the safety properties of P_h may not preserve liveness.
- When the liveness property $\mathcal{L}_h$ of P_h needs to be preserved by program P_l with a liveness property $\mathcal{L}_l$, we require that

$$\exists \overline{x}.(\Pi(P_l) \wedge \mathcal{L}_l) \Rightarrow \exists \overline{y}.(\Pi(P_h) \wedge \mathcal{L}_h)$$

- To prove this implication with respect to a refinement mapping, we need to prove the *liveness preservation* as well:

$$\Pi(P_l) \wedge \mathcal{L}_l \Rightarrow \widetilde{\mathcal{L}_h}$$

- **Liveness is more difficult to deal with**
- **Safety is more important**

Completeness

- The validity of $\Phi(P_l) \Rightarrow \Phi(P_h)$ does not imply the existence of a refinement mapping.
- In general however, a refinement mapping can be found by adding dummy variables to specifications.
- Once a refinement mapping is found, the verification of the refinement is straightforward and can be aided by mechanical means.
- Finding a refinement mapping may be difficult if it is not known how P_l is obtained from P_h. However, knowing how an abstract state variable in P_h is implemented by the variables in P_l, it is not difficult 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.

FAULT-TOLERANCE

Overview

Spec.		*Prog.*	*Impl.*
S	$\xleftarrow{\text{Sat}}$	P	**No faults**
?	$\xleftarrow{\text{Sat}}$	P	**Faults**

P is fault-tolerant if

S	$\xleftarrow{\text{Sat}}$	P	**Faults**

Problems

- How to formally define the fault-tolerance of a program?
- How to formally verify the fault-tolerance of a program?
- How to construct a fault-tolerant program?

A Solution

Specification		Program	Implementation
S	*Is Satisfied By*	P	**No Fault :-)**
?	*Is Satisfied By*	P	F **:-(**
?	*Is Satisfied By*	$\mathcal{F}(P, F)$	**No Fault :-(**
S	*Is Satisfied By*	$\mathcal{F}(P_{FT}, F)$	**No Fault :-)**

Notions

- F is an interfering 'program' representing the fault-environment.
- $\mathcal{F}$ is the fault-transformation.
- P_{FT} is a F-tolerant implementation of S.

Modelling Faults and Fault-Tolerance

- *Faults* are modelled as actions which change states.
- The *effect* of a set of faults F on a program P is modelled by a transformation:

  $\mathcal{F}(P, F) \stackrel{def}{=} \langle \overline{v}, \Theta, A \cup F \rangle$ F-affected version of P

- A program P is a *F-tolerant implementation* of a specification ψ iff

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

- A program P_l is a *F-tolerant refinement* of P_h iff

  $\mathcal{F}(P_l, F) \sqsupseteq P_h$

Example Continued

- Fault-affected program

$$
\begin{array}{lcl}
fault & \stackrel{def}{=} & d' \neq d \\
F_1 & \stackrel{def}{=} & \{fault\} \\
\mathcal{F}(P_1, F_1) & = & \langle \overline{v}_1, \overline{x}_1, \Theta_1, \{R_1^p, W_1^p, R_1^m, W_1^m, fault\} \rangle \\
\mathcal{N}_{\mathcal{F}(P_1, F_1)} & = & \mathcal{N}_{P_1} \vee fault \\
\Pi(\mathcal{F}(P_1, F_1)) & = & \Theta_1 \wedge \Box[\mathcal{N}_{\mathcal{F}(P_1, F_1)}]_{\overline{v}_1} \\
\Phi(\mathcal{F}(P_1, F_1)) & = & \exists d. \Pi(\mathcal{F}(P_1, F_1))
\end{array}
$$

- It is easy to see that

$$\Phi(\mathcal{F}(P_1, F_1)) \not\Rightarrow \Phi(P_1) \quad OR \quad \mathcal{F}(P_1, F_1) \not\sqsupseteq P_1$$

- In particular, $\mathcal{F}(P_1, F_1)$ does not satisfies

$$\exists d. \Box((op \neq w) \Rightarrow (d = val))$$

Example Continued

Triplicate memory and assume that at any time at most one memory may fail, and let

$$vote(x, y, z) \stackrel{def}{=} \textbf{if } x = y \textbf{ then } x \textbf{ else } z$$

$$\begin{aligned}
\overline{v}_2 &\stackrel{def}{=} \{op, val, d_1, d_2, d_3, f_1, f_2, f_3\} \\
\overline{x}_2 &\stackrel{def}{=} \{d_1, d_2, d_3, f_1, f_2, f_3\} \\
\Theta_2 &\stackrel{def}{=} (op = w) \wedge val \in \mathbf{Z} \wedge (f_i = 0) \wedge (d_i = d_j) \\
P_2 &\stackrel{def}{=} \langle \overline{v}_2, \overline{x}_2, \Theta_1, \{R_2^p, W_2^p, R_2^m, W_2^m\} \rangle \\
fa_i &\stackrel{def}{=} d_i' \neq d_i \wedge f_i' = 1 \\
F_2 &\stackrel{def}{=} \{fa_i : i = 1, 2, 3\}
\end{aligned}$$

Example Continued - Specification

$$\mathcal{N}_{F_2} = fa_1 \vee fa_2 \vee fa_3$$

$$\mathcal{B}_{F_2} \stackrel{def}{=} \Box(f_1 + f_2 + f_3 \leq 1)$$

$$R_2^p \stackrel{def}{=} (op = rdy) \wedge (op' = r)$$

$$W_2^p \stackrel{def}{=} (op = rdy) \wedge (op' = w) \wedge (val' \in \mathbf{Z})$$

$$R_2^m \stackrel{def}{=} (op = r) \wedge (op' = rdy) \wedge (val' = vote(d_1, d_2, d_3))$$

$$W_2^m \stackrel{def}{=} (op = w) \wedge (op' = rdy) \wedge \forall i.(d_i' = val \wedge f_i' = 0)$$

Example Continued

$$\mathcal{N}_{\mathcal{F}(P_2,F_2)} = \mathcal{N}_{P_2} \vee \mathcal{N}_{F_2}$$

$$\Pi(\mathcal{F}(P_2, F_2)) = \Theta_2 \wedge \Box[\mathcal{N}_{\mathcal{F}(P_2,F_2)}]_{\overline{v}_2} \wedge \mathcal{B}_{F_2}$$

$$\Phi(\mathcal{F}(P_2, F_2)) = \exists \overline{x}_2 . \Pi(\mathcal{F}(P_2, F_2))$$

Fault-Tolerance = Refinement

Theorem: Under the assumption $\mathcal{B}_{F_2}$

$$\mathcal{F}(P_2, F_2) \sqsupseteq P_1 \text{ (or } P_2 \sqsupseteq_{F_2} P_1)$$

Define a refinement mapping?

Proof

- Define the mapping from the states of $\overline{v}_2$ to those of d:
  $\widetilde{d} = vote(d_1, d_2, d_3)$
- Then prove the step-simulation from:
  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
  $(op = w) \wedge (op' = rdy) \wedge (vote(d_1', d_2, d_3') = val)$
  Case 3: No fa_i-step changes the values val and op, thus it is sufficient to show that no fa_i-step changes $\tilde{d}$: for $i = 1$, $\mathcal{B}_{F_2} \wedge \Pi(\mathcal{F}(P_2, F_2))$ implies
  - $\Box(f_1 = 1 \Rightarrow (d_2 = d_3))$
  - $fault_1 \Rightarrow f_1' = 1$, $fault_1 \Rightarrow (d_2' = d_3')$
  - $(d_2' = d_3') \Rightarrow unchanged(\tilde{d})$

Real-Time

Make the clock display the "*correct real time*".

- Use a observable variable, *now* to represent time
- Assume the change of the display is instantaneously

$$\begin{bmatrix} hr \mapsto 12, \\ now \mapsto \sqrt{2.47} \end{bmatrix}, \begin{bmatrix} hr \mapsto 12, \\ now \mapsto \sqrt{2.5} \end{bmatrix}, \ldots,$$

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

$$tNxt(HCnxt) \stackrel{def}{=} (t' = 0) \lhd HCnxt \rhd (t' = t + (now' - now))$$

$$Timer(t, HCnxt) \stackrel{def}{=} (t = 0) \wedge \Box[tNxt]_{(t,hr,now)}$$

Time Bounds

- $Max(t, 360 + \rho) \stackrel{def}{=} \Box(t \leq 360 + \rho)$
- $Min(t, HCnxt, 360 - \rho) \stackrel{def}{=} \Box[HCnxt \Rightarrow (t \geq 360 - \rho)]_{hr}$

  $Time(HCnxt) \stackrel{def}{=} Timer(t, HCnxt) \wedge Max \wedge Min$
- $RTHC \Rightarrow HC \wedge Time(HCnxt, t)$
- What is more needed?

NB: $Timer(t, HCnxt)$ is a *counting up* timer

Specification of Real-Time

- Time starts from 0: $\mathit{NowInit} \stackrel{def}{=} now = 0$
- Time advance:
  $\mathit{NowNxt} \stackrel{def}{=} now' \in \{r \in \mathbf{R}^{\geq 0} : r > now\} \wedge (hr' = hr)$
- Time Divergence: $NZ \stackrel{def}{=} \forall t \in \mathbf{R}^{\geq 0} \cdot \Diamond(now > r)$
- $RT \stackrel{def}{=} \mathit{NowInit} \wedge \Box[\mathit{NowNxt}]_{now} \wedge NZ$
- The Real-Time Clock: $\mathit{RTHC} \stackrel{def}{=} \exists t \cdot \mathit{HC} \wedge \mathit{Time}(\mathit{HCnxt}) \wedge \mathit{RT}$

Healthiness Conditions of RT

Timed Transition Systems

- Time domain: $\mathbf{R}^{\geq 0}$
- Program $P = (\overline{v}, \overline{x}, \Theta, A)$
- A timed version of P: timed version P^T of P is tuple $P^T = (P, L, U)$, where L and U are functions from A to $\mathbf{R}^{\geq 0} \cup \{\infty\}$, such that

  $L(\tau) \leq U(\tau)$ for every $\tau \in A$
- A timed state of P^T is a pair (s, t).
- A timed computation of P^T is an infinite sequence of timed states $\rho = (\sigma_0, t_0), (\sigma_1, t_1), \ldots, ..$ such that

  $$RT \wedge \bigwedge_{\tau \in A} \textbf{\textit{Time}}(\tau, t_\tau)$$
- **Semantics:** $[\![P^T]\!]$ is the set of all timed computations of P^T.
- **Refinement:** $P^T \sqsupseteq Q^T$ iff $[\![P^T]\!] \subseteq [\![Q^T]\!]$

Volatile Time Bounds

For each operation τ, assign bounds:

- perform τ only if it has been continuously enabled for lower bound $L(\tau)$
- τ must not be continuously enabled for upper bound $U(\tau)$ without being performed.

So a real-time program, P^T can be represented as

$$P^T = \langle P, L, U \rangle \text{ where } L(\tau) \leq U(\tau)$$

Specification of a Real-Time Program

- The timing bounds for an operation τ can be specified as TLA formulas LB_τ and UB_τ with:

$$LB_\tau = true \text{ if } L(\tau) = 0$$

$$UB_\tau = true \text{ if } U(\tau) = \infty$$

- The internal and external specifications of P^T:

$$\Pi(P^T) \stackrel{def}{=} \Pi(P) \land RT \land B_P$$

$$\Phi(P^T) \stackrel{def}{=} \exists \overline{x}.\Pi(P^T)$$

where $\overline{x}$ are the internal variables including timers.

Logical Definition of Timers

Let $\tau \in A$ and $\delta \in \mathcal{R}^+ \cup \{\infty\}$,

- Counting Up Timer

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

- Specify the time bounds

$$\mathit{Max} \uparrow (\tau) \stackrel{def}{=} \Box(t_\tau \leq U(\tau))$$

$$\mathit{Min} \uparrow (\tau) \stackrel{def}{=} \Box[\tau \Rightarrow t \geq L(\tau)]_{(\overline{v}, now)}$$

$$B_\tau \uparrow \stackrel{def}{=} \mathit{Timer}(t_\tau, \tau) \wedge \mathit{Min} \uparrow (\tau) \wedge \mathit{Max} \uparrow (\tau)$$

$$B_P \uparrow \stackrel{def}{=} \bigwedge_{\tau \in A} B_\tau$$

Counting Down Timer

$$\mathit{Volatile}(\tau, t, \delta, \overline{v}) \stackrel{def}{=}$$

$$((en(\tau) \wedge t = \delta) \vee (\neg en(\tau) \wedge t = \infty)) \wedge$$

$$\Box[(\quad en(\tau)' \wedge (\tau \vee \neg en(\tau)) \wedge t' = now' + \delta$$

$$\vee\; en(\tau) \wedge en(\tau)' \wedge \neg\tau \wedge t' = t$$

$$\vee\; \neg en(\tau)' \wedge t' = \infty) \wedge (\overline{v}' \neq \overline{v})]_{(t, \overline{v}, now)}$$

Specification of Time Bounds

Define

$$Min(\tau, t, \overline{v}) \stackrel{def}{=} \Box[\tau \Rightarrow (t \leq now)]_{(\overline{v}, now)}$$

$$Max(t) \stackrel{def}{=} \Box[now' \leq t]_{now}$$

$$LB_\tau \stackrel{def}{=} Volatile(\tau, t_\tau, L(\tau), \overline{v}) \land Min(\tau, t, \overline{v})$$

$$UB_\tau \stackrel{def}{=} Volatile(\tau, T_\tau, U(\tau), \overline{v}) \land Max(\tau)$$

$$B_P \stackrel{def}{=} \bigwedge_{\tau \in A} LB_\tau \land UB_\tau$$

Can you prove that a program with counting up timers are equivalent to its counterpart with counting down timers?

The Main 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,
- the processor periodically issues an operation,
- the period ρ of issuing an operation must be greater than the upper bound for the memory to execute the operation.

The Main Example Continued

The real-time program $P_1^T = \langle P_1, L_1, U_1 \rangle$ is described as follows:

$$
\begin{array}{rcl}
\overline{v}_1 & \stackrel{def}{=} & \{op, val, c, d\},\ \overline{c}_1 \stackrel{def}{=} \{c, d\} \\
\Theta_1 & \stackrel{def}{=} & (op = w) \wedge (val \in \mathbf{Z}) \wedge \neg c \\
RW_1^p & \stackrel{def}{=} & (op' = r) \wedge \neg c' \vee (op' = w) \wedge \neg c' \wedge (val' \in \mathbf{Z}) \\
R_1^m & \stackrel{def}{=} & (op = r) \wedge \neg c \wedge (val' = d) \wedge c' \\
W_1^m & \stackrel{def}{=} & (op = w) \wedge \neg c \wedge (d' = val) \wedge c' \\
A_1 & \stackrel{def}{=} & \{RW_1^p, R_1^m, W_1^m\} \\
L_1(RW_1^p) & = & U_1(RW_1^p) = \rho \\
L_1(R_1^m) & = & L_1(W_1^m) = 0 \\
U_1(R_1^m) & = & U_1(W_1^m) = D_1 < \rho \\
P_1^T & = & \langle \overline{v}_1, \Theta_1, A_1, L_1, U_1 \rangle
\end{array}
$$

Example Continued

The specification of the real-time interface:

$$
\begin{aligned}
LB_1 = \quad & Volatile(t_{RW_1^p}, RW_1^p, \rho, \overline{v}_1) \wedge Min(t_{RW_1^p}, RW_1^p, \overline{v}_1) \\
UB_1 = \quad & Volatile(T_{RW_1^p}, RW_1^p, \rho, \overline{v}_1) \wedge \\
& Volatile(T_{R_1^m}, R_1^m, D_1, \overline{v}_1) \wedge \\
& Volatile(T_{W_1^m}, W_1^m, D_1, \overline{v}_1) \wedge \\
& Max(T_{RW_1^p}) \wedge Max(T_{R_1^m}) \wedge Max(T_{W_1^m})
\end{aligned}
$$

Then

$$\Pi(P_1^T) = \quad \Theta_1 \wedge \Box[RW_1^p \vee R_1^m \vee W_1^m]_{\overline{v}_1} \wedge RT \wedge LB_1 \wedge UB_1$$

$$\Phi(P_1^T) = \exists(t_{RW_1^p}, T_{RW_1^p}, T_{R_1^m}, T_{W_1^m}, c, d).\Pi(P_1^T)$$

Verifying and Refining a Real-Time Program

- *Verification:* P^T implements a specification ψ iff

  $$\Phi(P^T) \Rightarrow \psi$$

- For example, define $\varphi \overset{\delta}{\leadsto} \psi$ as

  $\forall t.\Box(\varphi \wedge now = t \Rightarrow \Diamond(\psi \wedge now \leq t + \delta))$

  Then

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

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

- *Refinement:* P_l^T refines P_h^T iff

  $$\Phi(P_l^T) \Rightarrow \Phi(P_h^T)$$

Proving a Refinement

1. Convert the exact specification of P^T at each side of the implication:

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

   into the form $\Theta \wedge \Box[\mathcal{N}]_{\overline{z}} \wedge NZ$, with

   $$NZ \stackrel{def}{=} \forall t \in \mathcal{R}^+ . \Diamond(now > t),$$ where

   - $\overline{z}$ equals $\overline{v}$ plus now and the timers;
   - Θ is obtained from Θ_P by conjoining it with the initial conditions on now and the timers;
   - $\mathcal{N}$ is an action formula.

2. Find a refinement mapping from the state space of P_l^T, including those of the timers, to the state space of P_h^T.

3. Check the initiality-preservation and step-simulation

BOTH FAULT-TOLERANCE AND REAL-TIME

- Fault-tolerant systems often also have real-time constraints, or real-time systems also often have fault-tolerant requirements.
- It is important that the timing properties of a program are refined along with the fault-tolerant and functional properties defined in the program specification.
- Fault-tolerant redundant actions, such as checkpointing operations and recovery operations, are also timed.
- The rate of occurrences of faults should be limited to achieve finite progress in the execution of the underlying program.

Fault-Tolerance in Real-Time Programs

- The functional assumption on faults is still modelled by a set of atomic actions F.
- No time bounds are imposed on fault operations – lower bound is 0 and upper bound is ∞.
- We need timing assumptions on the occurrence of faults are conjunctions of the form

  $$\Box(fa_1 \Rightarrow \Box_\varepsilon \neg fa_2)$$

  whenever fa_1 *occurs,* fa_2 *cannot occur within* ε *units of time.*

  ε should be long enough so that the recovery can take place and progress can be made.

Verification of Real-Time and Fault-Tolerance

- The fault-affected version of P^T is

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

- The specifications of $\mathcal{F}(P^T, F)$ are

$$\Pi(\mathcal{F}(P^T, F)) \stackrel{def}{=} \Pi(\mathcal{F}(P, F)) \wedge B_P \wedge \mathcal{T}_F$$

$$\Phi(\mathcal{F}(P^T, F)) \stackrel{def}{=} \exists \overline{x}.\Pi(\mathcal{F}(P^T, F))$$

- P^T is a F-tolerant implementation of a specification ψ iff

$$\Phi(\mathcal{F}(P^T, F)) \Rightarrow \Psi$$

- P_l^T is a F-tolerant refinement of P_h^T iff

$$\Phi(\mathcal{F}(P_l^T, F)) \Rightarrow \Phi(P_h^T)$$

Example Continued

The real-time version of P_2

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

The Time Bounds Required

Meeting the timing properties of P_1^T requires

$$L_2(RW_2^p) = U_2(RW_2^p) = \rho$$

$$L_2(R_2^m) = L_2(W_2^m) = 0$$

$$U_2(R_2^m) = U_2(W_2^m) = D_2 \leq D_1$$

Real-Time fault-Tolerance

Theorem: $P_2^T \sqsupseteq_{F_2} P_1^T$ under the assumption $\mathcal{B}_{F_2}$.

Define a refinement mapping from the states of P_2^T's to the states over the internal variables of P_1^T:

$$\tilde{d} \stackrel{def}{=} vote(d_1, d_2, d_3) \qquad \tilde{c} \stackrel{def}{=} c$$

$$\widetilde{t_{RW_1^p}} \stackrel{def}{=} t_{RW_2^p} \qquad \widetilde{T_{RW_1^p}} \stackrel{def}{=} T_{RW_2^p}$$

$$\widetilde{T_{R_1^m}} \stackrel{def}{=} T_{R_2^m} \qquad \widetilde{T_{W_1^m}} \stackrel{def}{=} T_{W_2^m}$$

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

Remarks

To achieve fault-tolerance with timing constraints,

- a more powerful (or faster) machine is often needed,
- with a machine of the same speed, the original time bounds must have enough slack to accommodate the redundant actions for fault-tolerance.

Example Continued

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 P_3

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

Example Continued – the Time Bounds

To meet timing properties of P_2^T, it is required

1. that the period for the processor to issue an operation is still ρ:

   $L_3(RW_3^p) = U_3(RW_3^p) = \rho$

2. that the upper bound D_{w_i} for the ith memory to execute an issued write is not greater than D_2:

   $U_3(W_3^{m_i}) = D_{w_i}, D_{w_i} \leq D_2$

3. 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 $i = 1, 2, 3$:

   $U_3(R_2^{m_i}) = D_{r_i}, U_3(Vote) = D_{vote}, D_{r_i} + D_{vote} \leq D_2$

4. and $L_3(R_3^{m_i}) = L_3(W_3^{m_i}) = L_3(Vote) = 0$

Example Continued – Correctness

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

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

Example Continued – the Refinement Mapping

A refinement mapping is given as follows;

$$\tilde{c} \stackrel{def}{=} c_1 \wedge c_2 \wedge c_3$$

$$\tilde{d_i} \stackrel{def}{=} \begin{cases} d_i & \text{if } c_1 \wedge c_2 \wedge c_3 \vee \neg c_1 \wedge \neg c_2 \wedge \neg c_3 \\ val & \text{otherwise} \end{cases}$$

$$\widetilde{T_{R_2^m}} \stackrel{def}{=} \begin{cases} min\{T_{R_3^{m_i}}\} + D_{vote} & \text{if } \vee_{i=1}^{3}(op_i = r \wedge \neg v_i) \\ T_{Vote} & \text{if } v_1 \wedge v_2 \wedge v_3 \\ \infty & \text{otherwise} \end{cases}$$

$$\widetilde{T_{W_2^m}} \stackrel{def}{=} min\{T_{W_3^{m_i}} : i = 1, 2, 3\}$$

Example Continued – Proof Outline

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 stepwise:

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.

Schedulability

Given:

- a program P^T with its operation set A partitioned into n processes

$$P^T = \langle\langle\overline{v}, I, p_1 \cup \ldots \cup p_n\rangle, L, U\rangle$$

- a description of a scheduler (its policy) S,
- a system with a set Pr of m processors of known speed.

Questions:

- How can it be proved that P^T satisfies its timing constraints when the processors are shared between the processes by the scheduler S^T?
- What will happen if some of the processors may fail?

A Solution

- The program and the scheduler are specified separately, but in the same language, and scheduler is specified *generically*.
- A transformation is then used to combine these specifications for a given set Pr of processors

  $\mathcal{I}(P, S, Pr) \stackrel{def}{=} \langle \overline{v} \cup \overline{u}, I \wedge I_S, A_P \cup A_S \rangle$

  $\mathcal{I}(P^T, S^T, Pr) \stackrel{def}{=} \langle \mathcal{I}(P, S, Pr), l_p \cup l_S, u_p \cup u_S \rangle$

  - For each $\tau \in p_i$, the corresponding operation in A_P is $run_i \wedge \tau$.
  - Time bounds of scheduling operations are defined by l_S and u_S.
  - Time bounds of the program operations are determined by l_p and u_p which represent the speed of the processors.

Feasibility

- The specifications of the implementation

$$\Pi(\mathcal{I}(P^T, S^T, Pr)) = \Pi(S^T) \wedge \Pi(P^T)$$

$$\Phi(\mathcal{I}(P^T, S^T, Pr)) = \exists \overline{u}.\Phi(S^T) \wedge \Phi(P^T)$$

- The implementation $\mathcal{I}(P^T, S^T, Pr)$ is *feasible* for a specification ψ iff

$$\Phi(\mathcal{I}(P^T, S^T, Pr)) \Rightarrow \psi$$

- The implementation $\mathcal{I}(P^T, S^T, Pr)$ is *feasible* iff

$$\Phi(\mathcal{I}(P^T, S^T, Pr)) \Rightarrow \Phi(P^T)$$

- Feasibility can be proved by proving some program invariants.

Untimed Scheduling

- A scheduler allocates a process of P for execution by a processor using a *submit action*.
- It removes a process from a processor by a *retrieve action*.
- 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.

Specifying Untimed Scheduling

- Let 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 τ of P in p_i is transformed as:
- Let $r(\tau)$ denote the transformed action of τ in $\mathcal{G}(P)$:

$$r(\tau) \stackrel{def}{=} run_i \wedge \tau$$

- Therefore, $en(r(\tau)) \Leftrightarrow run_i \wedge en(\tau)$.

Untimed Scheduler

- A scheduler described as an untimed program S:
  - whose submit and retrieve actions modify run_i,
  - whose initial condition $idle \stackrel{def}{=} \forall i. \neg run_i$.
- We use a *generic* parameterized description $S(n, m)$ for a scheduler so that it can be applied to a P consisting of any number n of processes on a system with any number m of processors.
- Therefore, a program P of n processes will be implemented by a scheduler $S(n, m)$ for some positive integer m.
- Given $S(n, m)$, the scheduling of P by $S(n, m)$ on a set of m processors can be described as a transformation $\mathcal{I}(P, S(n, m))$.
  - its initial condition is $idle \wedge \Theta$,
  - its actions are formed by the union of the actions of $S(n, m)$ and $\mathcal{G}(P)$ and their execution is interleaved.

Execution of a Scheduled Program

An **execution** of $\mathcal{I}(P, S(n, m))$ is a state sequence σ over the union of the variables $\overline{z}$ of S and the variables $\overline{v}$ of P for which,

1. the initial state σ_0 satisfies the initial conditions Θ of P and $idle$ of $S(n, m)$,
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 $S(n, m)$, 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 Specification of a Scheduled Program

- $\mathcal{I}(P, S(n,m))$ is then specified by

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

- We assume that $S(n,m)$ does not *change* the state of P, i.e.

$$\mathcal{N}_{S(n,m)} \Rightarrow (\overline{v}' = \overline{v})$$

- This gives us the compositional specification

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

- It can be seen that $r(\tau) \Rightarrow \tau$ holds for each action τ of P.
- Thus $\Pi(\mathcal{G}(P)) \Rightarrow \Pi(P)$.

Scheduling = Refining

Theorem: Given a program P of n processes, for any positive integer m, we have

$$\Pi(\mathcal{I}(P, S(n,m))) \Rightarrow \Pi(P)$$

This shows that $\mathcal{I}(P, S(n,m))$ refines P and the transformation $\mathcal{I}$ (and thus the scheduler $S(n,m)$) preserves the functional properties of P.

Timed Scheduling

- The timing properties of $\mathcal{I}(P, S(n, m))$ depend on the number m 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 $[l(\tau), u(\tau)]$.
- The functions l and u define the (persistent) time bounds of the actions in $\mathcal{G}(P)$.
- The real-time program

$$\mathcal{G}(P)^T \stackrel{def}{=} \langle \mathcal{G}(P), l, u \rangle$$

where for each $r(\tau)$ of $\mathcal{G}(P)$, $l(r(\tau)) = l(\tau)$ and $u(r(\tau)) = u(\tau)$.

Schedule a Real-Time Program

- To guarantee that P^T satisfies its real-time deadlines, the computational overhead of the *submit* and *retrieve* actions must be bounded.
- Let the scheduler $S(n, m)$ 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** The implementation of P^T under $S(n, m)^T$ is the composition of $\mathcal{G}(P)^T$ and $S(n, m)^T$ and defined as follows:

$$\mathcal{I}(P^T, S(n, m)^T) \stackrel{def}{=} \langle \mathcal{I}(P, S(n, m)), L_{\mathcal{I}(P)}, U_{\mathcal{I}(P)} \rangle$$

  where the functions $L_{\mathcal{I}(P)}$ and $U_{\mathcal{I}(P)}$ are respectively the union L_S and l, and the union of U_S and u.
- Thus the execution speed of the processors and the timing properties of the scheduler determine the timing properties of the scheduled program.

Persistent Timer

Given a P^T of n processes $p_1, \ldots, p_n$, let $\delta \in \mathbf{R}^+$, and τ be an action in p_i.

We define a **persistent δ-timer t for an action** τ as:

$$
Persistent(t, \tau, \delta, \overline{v}) \stackrel{def}{=} t = \delta \wedge
$$

$$
\Box \left[\begin{array}{lll}
(& r(\tau) \wedge t' = now + \delta & \text{taken} \\
\vee & en(r(\tau)) \wedge \neg r(\tau) \wedge t' = t & \text{running} \\
\vee & \neg en(\tau)' \wedge t' = now' + \delta & \text{disabled} \\
\vee & en(\tau) \wedge \neg run_i \wedge & \\
 & t' = t + (now' - now)) & \text{pre-empted} \\
\wedge & ((\overline{v}, now)' \neq (\overline{v}, now)) &
\end{array} \right]_{(t, \overline{v}, now)}
$$

Specification of an Implementation

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

$$B(\mathcal{G}(P)^T) \stackrel{def}{=}$$
$$\bigwedge_{\tau \in A} Persistent(t_\tau, \tau, l(\tau), \overline{v}) \wedge Min(t_\tau, r(\tau), \overline{v}) \wedge$$
$$\bigwedge_{\tau \in A} Persistent(T_\tau, \tau, u(\tau), \overline{v}) \wedge Max(T_\tau)$$

- Hence, the exact specification of $\mathcal{I}(P^T)$ is

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

The Normal Specification

$$\Phi(\mathcal{I}(P^T, S(n,m)^T)) = \exists \overline{z}.(\Phi(S(n,m)^T) \wedge \Phi(\mathcal{G}(P)^T)) \tag{1}$$

Verification of an Implementation

Lemma 1: Let P^T be a RT program with n processes, $S(n,m)^T$ a RT scheduler, φ a TLA formula, and ψ a TLA formula which does not contain free state variables in $\overline{z}$.

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

Proof

1. Since $\Phi(S(n,m)^T) \Rightarrow \varphi$, we have

$$\Phi(S(n,m)^T) \wedge \Phi(\mathcal{G}(P)^T) \Rightarrow \varphi \wedge \Phi(\mathcal{G}(P)^T)$$

2. By Premise 2 in the rule, we have $\varphi \wedge \Phi(\mathcal{G}(P)^T) \Rightarrow \psi$
3. Thus, we have $\Phi(S(n,m)^T) \wedge \Phi(\mathcal{G}(P)^T) \Rightarrow \psi$
4. As ψ does not contain free variables in $\overline{z}$, we have

$$\exists \overline{z}.(\Phi(S(n,m)^T) \wedge \Phi(\mathcal{G}(P)^T)) \Rightarrow \psi$$

5. From Equation 1, we have

$$\Phi(\mathcal{I}(P^T, S(n,m)^T)) \Rightarrow \psi$$

Feasibility

- **Definition:** The timed scheduled program $\mathcal{I}(P^T, S(n,m)^T)$ is *feasible* if $\Phi(\mathcal{I}(P^T, S(n,m)^T)) \Rightarrow \Phi(P^T)$ is valid.

- To prove the feasibility, it is sufficient to find a refinement mapping such that:

$$\Pi(\mathcal{I}(P^T, S(n,m)^T)) \Rightarrow \widetilde{\Pi(P^T)}$$

- Assume that $\Phi(S^T) \Rightarrow \varphi$, the feasibility can be proved from Rule R1 as

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

- Find a refinement mapping from the states of $\overline{x} \cup \overline{z} \cup timer(\mathcal{G}(P^T))$ to the states of $\overline{x} \cup time(P^T)$, and then prove the implication

$$\varphi \wedge \Phi(\mathcal{G}(P)^T) \Rightarrow \widetilde{\Pi(P^T)} \qquad (3)$$

Find a Refinement Mapping

1. Introduce dummy timers into $\mathcal{I}(P^T, S(n,m)^T)$ corresponding to the timers of P^T.

$$dummies \stackrel{def}{=} \{h_\tau, H_\tau \mid \tau \in A\}$$

which are defined by

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

2. **Lemma 2:** Implication (2) is equivalent to

$$\exists dummies, \overline{z}.\varphi \wedge \Phi(\mathcal{G}(P)^T) \wedge D(dummies) \Rightarrow \Phi(P^T) \quad (4)$$

3. Defining a mapping from $\overline{x} \cup \overline{z} \cup timer(\mathcal{G}(P)^T) \cup dummies$ to $\overline{x} \cup timer(P^T)$:

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

4. **Lemma 3:** Let $TimedSched \stackrel{def}{=} \varphi \wedge \Pi(\mathcal{G}(P)^T) \wedge D(dummies)$. Then implication (4) is equivalent to

$$TimedSched \Rightarrow \widetilde{\Pi(P^T)} \tag{5}$$

Theorem 2: Given a real-time program P^T, an implementation $\mathcal{I}(P^T, S(n,m)^T)$ by a real-time scheduler $S(n,m)^T$ that satisfies a property φ is feasible if and only if the following implication hold for each action τ of P:

$$TimedSched \Rightarrow Max(H_\tau) \wedge Min(h_\tau, \tau, \overline{v}) \tag{6}$$

Proof

1. Recall that $\widetilde{\Pi(P^T)} = \widetilde{\Pi(P)} \wedge \widetilde{RT} \wedge \widetilde{B(P^T)}$
2. Obviously, $\widetilde{RT} = RT$ and $\widetilde{\Pi(P)} = \Pi(P)$.
3. Also $\Pi(\mathcal{G}(P)^T)$ implies $\Pi(\mathcal{G}(P))$, which in turn implies $\Pi(P)$.
4. Therefore, $\widetilde{RT}$ and $\widetilde{\Pi(P)}$ can be discarded from the right hand side of (3), i.e. (5) holds iff the following implication holds:

   $$TimedSched \Rightarrow \widetilde{B(P^T)} \tag{7}$$

5. Furthermore,

   $$\widetilde{B(P^T)} = \ D(dummies) \wedge \bigwedge_{\tau \in A} (Max(H_\tau) \wedge Min(h_\tau, \tau, \overline{v}))$$

6. Since $D(dummies)$ appears on LHS of Implication (7), Implication (7) holds iff for each τ of P the following implication holds:

   $$TimedSched \Rightarrow Max(H_\tau) \wedge Min(h_\tau, \tau, \overline{v})$$

Invariants

- An invariant of a $Prog$ is a property of the form $\Box Q$, where Q is a state predicate.
- An invariant can be easily established from its canonical specification using the following rule:

$$R2.\quad \frac{\begin{array}{lll} 1 & \Theta \Rightarrow Q_1 & \text{Initially } Q_1 \text{ holds} \\ 2 & Q_1 \wedge \tau \Rightarrow Q_1' & \text{Each step of the transition preserves } Q \\ 3 & Q_1 \Rightarrow Q & \end{array}}{\Pi(Prog) \Rightarrow \Box Q}$$

- Notice that both $Max(H_\tau)$ and $Min(h_\tau, \tau, \overline{v})$ not *invariant properties*.

Proving Feasibility = Proving Invariants

Theorem 3: For the program, scheduler, and action τ in Theorem 2, Implication (6) holds iff the following two equations hold

$$TimedSched \Rightarrow \Box(en(\tau) \Rightarrow T_\tau \leq H_\tau) \tag{8}$$

$$TimedSched \Rightarrow \Box(en(\tau) \Rightarrow t_\tau \geq h_\tau) \tag{9}$$

Feasibility of FTRT Programs

- Faults do not depend on the scheduler, and the F-affected scheduled program of P^T by a scheduler S is modelled as $\mathcal{F}(\mathcal{I}(P^T), F)$
- Its exact specification $\Pi(\mathcal{F}(\mathcal{I}(P^T), F))$ equals

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

- Let $TimedSched$ be redefined as

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

- $\mathcal{I}(P^T)$ is F*-tolerantly feasible* if the following implication holds

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

- All the equations and rules remain valid for fault tolerant feasibility.
- Assume that $P^T = \langle P, L, U \rangle$ is a F-tolerant refinement of P_h^T. Then any F-tolerant feasible implementation of P^T is a F-tolerant refinement of P_h^T.

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.

A Scheduler Specification

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

Processor Speed

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

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

Feasibility Result

Then it can be proved using the rules 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.

Fixed Priority Scheduling with Pre-Emption

Consider program P with of n *independent* tasks

The environment E activates the tasks periodically, with ρ_i as the period of τ_i, for $i = 1, \ldots, n$

Θ	$\stackrel{def}{=}$	$(0 \leq inv_i \leq 1) \wedge (com_i = 0)$	
α	$\stackrel{def}{=}$	$inv_i' = inv_i + 1$	action of E for task invocation
τ_i	$\stackrel{def}{=}$	$inv_i > com_i \wedge com_i' = com_i + 1$	action of P for task completion
$L(\alpha_i)$	$=$	$U(\alpha_i) = \rho_i$	period of invocation
$L(\tau_i)$	$=$	0 and $U(\tau_i) = D_i$	deadline of task

Requirement: each invocation of a task is completed before its next invocation

Specification of the Requirement

$$\Phi(E^T) \wedge \Phi(P^T) \Rightarrow \bigwedge_{i=1}^{i=n} \Box(inv_i \geq com_i \geq inv_i - 1)$$

Specify the Scheduling Policy

- A single processor using a pre-emptive, fixed-priority scheduler
- No scheduling overhead.
- Let τ_i have a higher priority than τ_j if $i < j$.
- Let g_i denote $en(\tau_i)$, and hr_i assert that τ_i has the highest priority:

  $$g_i \stackrel{def}{=} inv_i > com_i \qquad hr_i \stackrel{def}{=} g_i \wedge \forall j < i.\neg g_j$$

  The scheduler $S^T = \langle S, L, U \rangle$:

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

- S^T ensures $\mathit{Valid} \stackrel{def}{=} \Box(i \neq j \Rightarrow \neg(run_i \wedge run_j))$

Feasibility Result

- 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 completion time R_i task τ_i can be defined by the equivalent recurrence relation.
  $(k+1)$th response time $R_i^{(k+1)}$ for process i is:

$$R_i^{(k+1)} = C_i + \sum_{j=1}^{i-1} \lceil \frac{R_i^{(k)}}{\rho_j} \rceil \times C_j \tag{10}$$

- $R_i^{(0)}$ is initially set to C_i

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

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

Conclusions

- Transformations and refinement can be used to develop fault-tolerant real-time programs, and for formal schedulability.
- No new semantics needed.
- Similar methods work for other models, e.g. CCS and CSP, probably with different complexity.
- Mechanised tools is essential for using this method for larger programs.
- We do not need to develop new tools for this purpose.
- A combination of scheduling theory into formal program development.
- General feasibility can be proven by proving invariants of the implementation.
- A clear interface between a program and a scheduler is provided by the transformation $\mathcal{I}$.

Exam Questions

1. Exercise 4 on page 178
2. Exercise 5 on pages 179-180
3. Exercise 7 on pages 191-192
4. Exercise 8 on page 215 if you are interested in an UNU-IIST Fellowship

Related Papers

1. Z. Liu and M. Joseph. *Transformation of Programs for Fault-Tolerance*, *Formal Aspects of Computing*, **4**(5), 1992.
2. Z. Liu and M. Joseph: *Specification and verification of recovery in asynchronous communicating systems*, in J. Vytopil (ed.), Formal Techniques in Real-Time and Fault-Tolerant Systems, Kluwer Academic Publishers, 1993.
3. Z. Liu and M. Joseph: Stepwise development of fault-tolerant reactive systems, in Proc.of FTRTFT'94, LNCS 863.
4. Z. Liu, M. Joseph and T. Janowski. Verification of schedulability For Real-Time Programs. *Formal Aspects of Computing*, **7**(5), pp.510-532, 1995
5. Z. Liu and M. Joseph: *Verification of Fault-tolerance and real-time*, *Proc. FTCS-26*, IEEE Computer Society Press, Sendai, Japan, June, 1996.
6. Z. Liu and M. Joseph: *Specification and verification of fault-tolerance, timing and scheduling*, *ACM Transactions on Languages and Systems*, **21**(1), 1999.
7. Z. Liu, *et al*. *Unifying Proof Methodologies of DC and LTL*. *Formal Aspects of Computing*, **16**(2), 2004.