Part 5 Real-Time Modeling

Timed Object-Z

- Real-time systems have both functional and timing requirements.
- Fundamental goal: Producing the correct result at the right time.
- Numerous approaches for using Z in the real-time specification:
  - Timed Refinement approach by Mahony and Hayes and ProCos approach by Ravn, He, Hoare *et al*.

    all variables are modelled as traces of values

    $$v : Time \rightarrow Value$$

  - Quartz approach by Fidge *et al*.

    a *now* variable is added to the specification to capture the current absolute time.
- How these techniques can be integrated with object-oriented methods?

Adding Real-Time to Object-Z

- $\mathbb{T} == \mathbb{N}$ (the absolute discrete time domain)
  $\mathbb{E} == \mathbb{P}_1\, \mathbb{T}$ (the possible execution times of each operation)
- every class have a secondary attribute '$now : \mathbb{T}$'
- synchronisation: $\forall\, o_1, o_2 : \mathbb{O} \bullet o_1.now = o_2.now$ (a global invariant)
- every operation implicitly include $now' \geq now$
- the time execution of the operation: $now' - now \in \mathbb{E}$

Framework for Timed Modeling in Object-Z

InputEnv

$supply : \mathbb{T} \pfun InputObjects$

...

...

SystemX

$inenv : InputEnv$

...

$Op_1 \mathrel{\widehat=} ...;\ Op_2 \mathrel{\widehat=} ...;\ ...;\ Op_n \mathrel{\widehat=} ...$

IdleTick

$\neg\ pre(op_1) \land \neg\ pre(op_2) \land ... \land \neg\ pre(op_n)$

$now' - now \in \{1\}$

$process \mathrel{\widehat=} (Op_1 \Box Op_2 \Box ... \Box Op_n \Box IdleTick);\ process$

Summary of Timed Object-Z Approach

- Integrated various Z approaches for specifying real-time requirement with object-oriented modelling techniques into a framework.
- Maintained the existing Object-Z semantics and style by not adding any new constructs into the specification language.
- Explicitly model time, no build-in timing construct.
- Single thread, poor in modeling true concurrent systems.
- Reference:

  J.S. Dong, J. Colton and L. Zucconi. A Formal Object Approach to Real-Time specification. In Proceedings of *APSEC'96*. IEEE Press, Dec 1996. (ISBN: 0818676388)

CSP/Timed CSP

- Hoare's CSP (Communicating Sequential Processes) an *event* based notation primarily aimed at de scribing the sequencing of behaviour within a process and the synchronisation of behaviour (or *communication*) between processes.
- Timed CSP extends CSP by introducing a capability to quantify temporal aspects of sequencing and synchronisation.

- S. Schneider. Concurrent and Real-time Systems: The CSP Approach, Wiley, 1999.
- A.W. Roscoe. The Theory and Practice of Concurrency. Prentice-Hall, 1997.
- J. Davies, Specification and Proof in Real-Time CSP, Cambridge University Press, 1993.
- C. A. R. Hoare. Communicating Sequential Processes. Prentice-Hall International, 1985.

Specifying a Process

A process is determined (specified) by what it can do;
i.e. a process is defined by its behaviour.
The perceived behaviour of a process will depend upon the observer.
We shall be mainly concerned with specifying the interaction between a system and its environment
(i.e. external (visible) behaviour).

Events

A process engages in *events*; each event is an atomic action. e.g. the events for a vending machine are

coin—insert a coin

choc—extract a chocolate

The set of events that a process can possibly engage in is the *alphabet* of the process e.g. the alphabet of the vending machine is

$$\{coin, choc\}$$

Traces

A *trace* is a finite sequence of events.
A (deterministic) process is specified by the set of traces denoting its possible behaviour. e.g. the traces of the vending machine:

$\langle\ \rangle$
$\langle coin \rangle$
$\langle coin, choc \rangle$
$\langle coin, choc, coin \rangle$
$\ldots$

Any execution of the process will be one of these sequences. If $s \frown t$ is a trace of a process,
then so also is s;
i.e. the set of traces is prefix closed.

Basic Process Notation

- lower case identifiers denote events;
  x, y, z are variables denoting events

- upper case identifiers denote processes;
  X, Y are variables denoting processes

- A, B, C denote sets of events

- if P is a process,
  αP denotes the alphabet of P

- if P is a process,
  $traces(P)$ denotes the set of traces of P

Trace Notation

- if A is a set of events,
  $\operatorname{seq} A$ denotes the set of all finite sequences of events from A.

- if $s, t : \operatorname{seq} A$,
  $s \frown t$ is the *concatenation* of s with t
  e.g.

$$\langle b, a, b \rangle \frown \langle b, c, a \rangle = \langle b, a, b, b, c, a \rangle$$

-

$$\begin{array}{|l} \leqslant : \operatorname{seq} A \leftrightarrow \operatorname{seq} A \\ \hline s \leqslant t \;\Leftrightarrow \\ \qquad \exists\, u : \operatorname{seq} A \bullet s \frown u = t \end{array}$$

- $s^n = s \frown s \frown s \frown \cdots \frown s$
  i.e. s concatenated with itself n times

Examples

(1) $STOP_A$ is the process with alphabet A that can do nothing.

$$traces(STOP_A) = \{\langle\,\rangle\}$$

(2) $CLOCK$ is the process with $\alpha CLOCK = \{tick\}$ which can 'tick' at any time.

$$traces(CLOCK) = tick^*$$

(3) VM is the process with $\alpha VM = \{coin, choc\}$ which repeatedly supplies a chocolate after a coin is inserted.

$$traces(VM) = \{s : \mathrm{seq}\{coin, choc\} \mid \exists\, n : \mathbb{N} \bullet s \leqslant \langle coin, choc\rangle^n\}$$

(4) $WALK$ is a one-dimensional random walk process with $\alpha WALK = \{left, right\}$.

$$traces(WALK) = (left \cup right)^*$$

(5) $LIFE$ is the process with $\alpha LIFE = \{beat\}$ which can stop (die) at any time.

$$traces(LIFE) = beat^*$$

Prefix

A process which may participate in event a then act according to process description P is written

$$a@t \rightarrow P(t).$$

The event a is initially enabled by the process and occurs as soon as it is requested by its environment, all other events are refused initially. The event a is sometimes referred to as the *guard* of the process. The (optional) timing parameter t records the time, relative to the start of the process, at which the event a occurs and allows the subsequent behaviour P to depend on its value. For examples:

$$VMU = coin \rightarrow STOP$$

$$SHORTLIFE = (beat \rightarrow (beat \rightarrow STOP)) = beat \rightarrow beat \rightarrow STOP$$

$$VMS = coin \rightarrow choc \rightarrow STOP$$

Understanding Timed Prefix

Let P be a process which has two free time variables t_1 and t_2. A possible execution of the prefix:

$a@t_1 \rightarrow b@t_2 \rightarrow P$

$\downarrow 3$ (time passed)

$a@t_1 \rightarrow b@t_2 \rightarrow P[(t_1 + 3)/t_1]$

$\downarrow a$ (event occur)

$b@t_2 \rightarrow P[3/t_1]$

$\downarrow 4$ (time passed)

$b@t_2 \rightarrow P[3/t_1][(t_2 + 4)/t_2]$

$\downarrow b$ (event occur)

$P[3/t_1][4/t_2]$

Other CSP/Timed-CSP primitives:

- $P;\ Q$ (sequential composition)
- $P\ |[\ X\ ]|\ Q$ (synchronous), $P\ |||\ Q$ (asynchronous)
- $a \to P\ \Box\ b \to Q$ (external choice), $a \to P \sqcap b \to Q$ (intern al choice)
- $P_1\ \triangledown\ e \to P_2$ (interrupt process)
- Wait $t;\ P$ (delay), $a \to P\ \triangleright\{t\}\ Q$ (time-out)

Sequential Composition

- The second form of sequencing is process sequencing. A distinguished event ✓ is used to represent and detect process termination.

- The sequential composition of P and Q, written $P;\ Q$, acts as P until P terminates by communicating ✓ and then proceeds to act as Q.

- The termination signal is hidden from the process environment and therefore occurs as soon as enabled by P. The process which may only terminate is written Skip.

Parallel composition

The parallel composition of processes P and Q, synchronised on event set X, is written

$$P \,|[\, X \,]|\, Q.$$

No event from X may occur in $P \,|[\, X \,]|\, Q$ unless enabled jointly by both P and Q. When events from X do occur, they occur in both P and Q simultaneously and are referred to as *synchronisations*. Events not from X may occur in either P or Q separately but not jointly. For example, in the process described by

$$(a \rightarrow P) \,|[\, a \,]|\, (c \rightarrow a \rightarrow Q)$$

all a events must be synchronisations between the two processes.

In an asynchronous parallel combination

$$P \,|||\, Q$$

both components P and Q execute concurrently without any synchronisations.

Choice

Diversity of behaviour is introduced through two choice operators.

The external choice operator allows a process a choice of behaviour according to what events are requested by its environment. The process

$$(a \to P) \,\Box\, (b \to Q)$$

begins with both a and b enabled. The environment chooses which event actually occurs by requested one or the other first. Subsequent behaviour is determined by the event which actually occurred, P after a and Q after b respectively.

Internal choice represents variation in behaviour determined by the internal state of the process. The process

$$a \to P \sqcap b \to Q$$

may initially enable either a, or b, or both, as it wishes, but must act subsequently according to which event actually occurred. The environment cannot affect internal choice.

Channel

A channel is a collection of events of the form $c.n$: the prefix c is called the *channel name* and the collection of suffixes is called the *values* of the channel.

When an event $c.n$ occurs it is said that *the value n is communicated on channel c*. When the value of a communication on a channel is determined by the environment (external choice) it is called an *input* and when it is determined by the internal state of the process (internal choice) it is called an *output*.

It is convenient to write $c?n : N \to P(n)$ to describe behaviour over a range of allowed inputs instead of the longer $\Box\, n : N \bullet c.n \to P(n)$. Similarly the notation $c!n : N \to P(n)$ is used instead of $\sqcap\, n : N \bullet c.n \to P(n)$ to represent a range of outputs.

e.g.

$$COPYBIT = in.0 \to out.0 \to COPYBIT \;\; \Box \;\; in.1 \to out.1 \to COPYBIT$$

$$\alpha COPYBIT = \{in.0, out.0, in.1, out.1\}$$

Interrupt

The interrupt process $P_1 \bigtriangledown e \to P_2$ behaves as P_1 until the first occurrence of interrupt event e, then the control passes to P_2.

Recursion

Recursion is used to given finite representations of non-terminating processes. The process expression

$$\mu\, P \bullet a?n : \mathbb{N} \to b!f(n) \to P$$

describes a process which repeatedly inputs a natural on channel a, calculates some function f of the input, and then outputs the result on channel b.

Traces of Processes

$$
\begin{aligned}
&traces(a \rightarrow P) = \\
&\qquad \{t : \operatorname{seq} A \mid t = \langle\,\rangle \\
&\qquad\qquad\qquad \lor \\
&\qquad\qquad\qquad \operatorname{head} t = a \ \land \ \operatorname{tail} t \in traces(P)\}
\end{aligned}
$$

$$
\begin{aligned}
&traces(a \rightarrow P \mid b \rightarrow Q) = \\
&\qquad traces(a \rightarrow P) \cup traces(b \rightarrow Q)
\end{aligned}
$$

$$
\begin{aligned}
&traces(x : B \rightarrow P(x)) = \\
&\quad \{t : \operatorname{seq} A \mid t = \langle\,\rangle \\
&\qquad\qquad \lor \\
&\qquad\qquad \operatorname{head} t \in B \ \land \ \operatorname{tail} t \in traces(P(\operatorname{head} t))\} \\
&\qquad = \cup\{x : B \bullet traces(x \rightarrow P(x))\}
\end{aligned}
$$

Traces of Recursive Process

$$traces(\mu\, X : A \bullet F(X)) = \\ \cup n : \mathbb{N} \bullet traces(F^n(STOP_A))$$

$$VM = \mu\, X : \{coin, choc\} \bullet (coin \rightarrow choc \rightarrow X)$$

so in this case

$$F(X) = coin \rightarrow choc \rightarrow X$$

and

$$F^0(STOP_A) = STOP_A$$
$$F(STOP_A) = coin \rightarrow choc \rightarrow STOP_A$$
$$F^2(STOP_A) \\ = F(F(STOP_A)) \\ = coin \rightarrow choc \rightarrow (coin \rightarrow choc \rightarrow STOP_A) \\ = coin \rightarrow choc \rightarrow coin \rightarrow choc \rightarrow STOP_A$$
$$F^3(STOP_A) = \cdots$$
$$\cdots$$

Timeout

The timeout construct passes control to an exception handler if no event has occurred in the primary process by some deadline.

The process

$$(a \to P) \triangleright \{t\}\ Q$$

will try to perform $a \to P$, but will pass control to Q if the a event has not occurred by time t, as measured from the invocation of the process. For example,

$$MayPrint1 = (receive \to print \to STOP) \triangleright \{60\}\ shutdown \to STOP$$

$$MP1(t) = (receive \to print \to STOP) \triangleright \{60 - t\}\ shutdown \to STOP$$

Exercise: Transmitter

A transmitter which repeatedly send a given message x until it receives and acknowledgement. Assume that the transmitter is in an environment which is always ready to accept a *send* message, then it will send the message every 5 time units until an *ack* message is received. (hint using recursion together with timeout).

Solution

$$Transmit(x) = send!x \rightarrow ((ack \rightarrow STOP) \triangleright\{5\}\ Transmit(x))$$

Delay

A process which allows no communications for period t then terminates is written $\textsc{Wait}\, t$. The process

$$\textsc{Wait}\, t;\ P \ = \ STOP \triangleright\{t\}\ P$$
$$a \xrightarrow{t} P \ = \ a \rightarrow \textsc{Wait}\, t;\ P \ = \ a \rightarrow (STOP \triangleright\{t\}\ P)$$

is used to represent P delayed by time t.

State parameters

In general, the behaviour of a process at any point in time may be dependent on its internal state and this may conceivably take an infinite range of values.

It is often not possible to provide a finite representation of a process without introducing some notation for representing this internal process state.

The approach adopted by CSP is to allow a process definition to be parameterised by state variables. Thus a definition of the form

$$P_{n:N} \mathrel{\widehat{=}} Q(n)$$

represents a (possibly infinite) family of definitions, one for each possible value of n.

There is no inherent notion of process state in CSP, but rather these annotations are a convenient way to provide a finite representation of an infinite family of process descriptions.

Exercise: A generic timed-collection

The generic timed-collection denotes a collection of elements of type X with a time stamp. Operations are allowed to add elements to and delete elements from the collection. When deleting an element from the collection, the oldest element should be removed and output to the element should be removed and output to the environment. The collection has the following timing properties. Firstly, that it updates the internal state during a *add* or *delete* operation. Secondly, each element of the collection becomes *stale* if it is not passed on within t_o time units of being added to the collection. Stale elements should never be passed on, but are instead purged from the collection upon becoming stale.

The generic function ps (purge stale) can be defined as

$$[X]$$

$$ps : (\mathbb{T} \times \mathbb{F}(\mathbb{T} \times X)) \to \mathbb{F}(\mathbb{T} \times X)$$

$$\forall\, t : \mathbb{T};\ s : \mathbb{F}(\mathbb{T} \times X) \bullet ps(t, s) = \{(t_o, e) : s \mid t_o > t \bullet (t_o - t, e)\}$$

e.g. $ps(2, \{(1, a), (3, b), (7, c)\}) = \{(1, b), (5, c)\}$.

Solution

$$TimedCollection \mathrel{\widehat{=}} TC_{\varnothing}.$$

$$TC_{\varnothing} \mathrel{\widehat{=}} left?e : X \rightarrow TC_{\{(t_o,e)\}}$$

$$TC_{\{(t,a)\}\cup s} \mathrel{\widehat{=}}$$
$$(left?e : X @ t_i \rightarrow TC_{ps(t_i,\{(t,a)\}\cup s)\cup\{(t_o,e)\}} \ \Box$$
$$right!a @ t_i \rightarrow TC_{ps(t_i,s)}) \rhd \{t\} \ TC_{ps(t,s)}$$

where $(t, a) = find\_oldest(\{(t, a)\} \cup s)$.

$[X]$

$$find\_oldest : \mathbb{P}_1(\mathbb{T} \times X) \rightarrow (\mathbb{T} \times X)$$

$$\forall s : \mathbb{P}_1(\mathbb{T} \times X) \bullet$$
$$\exists (t, e) : s \bullet t = min(\operatorname{dom} s)$$
$$find\_oldest(s) = (t, e)$$

Summary

For such an example Timed CSP is superior to Object-Z as a means of describing process control.

Timed CSP also handles the timing issues of delays and timeouts simply and elegantly. The allowed sequences of events are clearly and concisely determined by the CSP code, there is no need to calculate preconditions nor is any other form of deep reasoning required to understand the ways in which the timed-collection may evolve.

On the other hand, the syntactic treatment of internal state in the above is complex and unwielding, distracting strongly from the basically elegant treatment of the delay and timeout issues.

CSP still has no standard support for state modeling in the form of mathematical toolkits and libraries nor are there modular techniques for constructing and reasoning about complex internal state.