SOFTWARE VERIFICATION RESEARCH CENTRE

DEPARTMENT OF COMPUTER SCIENCE

THE UNIVERSITY OF QUEENSLAND

Queensland 4072
Australia

TECHNICAL REPORT

No. 95-1

An Action-Based Formal Model for Concurrent Real-Time Systems

C. J. Fidge and A. J. Wellings

September 1996
(Revision 1.1)

Phone: +61 7 3365 1003
Fax: +61 7 3365 1533

Note: Most SVRC technical reports are available via anonymous ftp, from `ftp.cs.uq.edu.au` in the directory `/pub/SVRC/techreports`.

Formal Aspects of Computing (1996) 3: 1–000
© 1996 BCS

An action-based formal model for concurrent real-time systems

C. J. Fidge[1] and A. J. Wellings[2]

[1]Software Verification Research Centre, Department of Computer Science, The University of Queensland, Queensland 4072, Australia; [2]Real-time Systems Research Group, Department of Computer Science, University of York, Heslington, York, Y01 5DD, United Kingdom.

Keywords: Real time; Concurrency; Action systems; Z; Ada 95.

Abstract. Action systems are a formalism for representing concurrent behaviours, based on interleaved atomic actions. We show how this model can be used to represent time-consuming, pre-emptible actions with real-time constraints. A development procedure is described which captures the steps programmers typically undertake in the design of real-time multi-tasking systems.

1. Introduction

Many approaches to modelling real-time systems have been proposed in recent years. However, for simplicity, most make unrealistic assumptions such as 'maximal parallelism' and 'instantaneous' actions. Such models are inadequate for representing the practical problems that programmers actually face when developing real-time systems using priority-driven, pre-emptive task scheduling.

Action systems [Bac92] are an attractive formalism for modelling concurrent systems. They extend the familiar notion of sequential state machines into the realm of concurrent systems by allowing independent atomic actions to be interleaved. Unfortunately the model has proven difficult to extend. Interleaving semantics makes it difficult to represent time-consuming parallel actions that may overlap in time. Action atomicity seems to preclude a model of pre-emption.

In this article we present a way of applying actions systems that *can* model such concepts. Absolute global time is modelled by a special 'now' variable, accessed by all processes. A global 'tick' action is used to effect the passage

Correspondence and offprint requests to: Colin Fidge, Department of Computer Science, The University of Queensland, Queensland 4072, Australia. E-mail: cjf@cs.uq.edu.au.

of time. Actions that consume time, and may be pre-empted, are modelled by deferring the observable effect of each action until it is known to have completed, in spite of other higher-priority actions that delay its progress. Application of the model is demonstrated by presenting a multi-layered development procedure for designing real-time multi-tasking systems.

2. Background: action systems

Action systems are a conservative extension of Back's refinement calculus [Bac92, BvW94]. They are intended for development of parallel and reactive programs, while still using familiar sequential refinement methods.

Syntactically, an action system $\mathcal{A}$ can be represented as an initialised **do** loop [Bac92].

$$
\begin{array}{l}
\mathcal{A} = \textbf{begin} \\
\quad \textbf{var}\ \vec{x} \bullet \\
\qquad A_0; \\
\qquad \textbf{do} \\
\qquad []_{i \in I}\ \mathbf{g}A_i \rightarrow \mathbf{s}A_i \\
\qquad \textbf{od} \\
\quad \textbf{end} : \vec{z}
\end{array}
$$

Here $\vec{x}$ denotes the *local variables* of $\mathcal{A}$ and $\vec{z}$ the *global variables* it accesses. Let $\vec{y} = \vec{x} \cup \vec{z}$. The *initialisation* predicate A_0 defines the initial state of the variables in $\vec{y}$ [BvW94]. The system itself consists of one or more *actions* A_i, where I is an indexing set. Each action A_i is a guarded command consisting of a *guard* $\mathbf{g}A_i$ and a *statement* $\mathbf{s}A_i$. Each guard is a predicate on variables in $\vec{y}$, defining the conditions under which the action can occur. Each statement is a predicate on before and after states, defining a non-deterministic multiple assignment to a subset of variables in $\vec{y}$. In this article we use the specification language Z [PST91] as a convenient, widely-known notation for representing these predicates. Also let $\vec{r}_i$ be the set of variables that action A_i only *reads*, $\vec{w}_i$ be those that it *writes*, and $\vec{a}_i$ be all those variables that it *accesses*. That is, $\vec{a}_i = \vec{r}_i \cup \vec{w}_i$ and $\vec{a}_i \subseteq \vec{y}$ [BS94].

Such a loop always has a well-defined meaning as a sequential program, of course. At each iteration one of the actions whose guard is true is nondeterministically chosen and executed, until no enabled action remains. However action systems may also be *interpreted* as describing a parallel system. In particular, in the *concurrent system* model, actions are logically grouped together to form *processes* [Bac92]. Where actions in different processes refer to the same variable that variable is *shared*. Actions must access shared variables under mutual exclusion and are thus considered atomic with respect to such variables. Actions that access disjoint parts of the state space are independent and may be thought of as occurring in either order or in parallel. Refinements must respect both the sequential and concurrent interpretations.

(Separate action system descriptions may also be composed using a '||' operator [Bac92]. However, as this is only a syntactic convenience for structuring specifications, we focus on a single action system specification here, with no loss of generality.)

Sequential systems can be semantically defined via the outputs they produce in response to each possible input. However, this is insufficient for parallel

systems that interact repeatedly with their environment. Action systems are therefore defined as having *behavioural semantics* [BvW94]. Each action system $\mathcal{A}$ is defined by the set $beh(\mathcal{A})$ of its *behaviours*, where a behaviour is a sequence s of *states* with components for both local and global variables.

$$s = \langle(\vec{x}_0, \vec{z}_0), (\vec{x}_1, \vec{z}_1), \ldots\rangle$$

Behaviours may be finite or infinite. Finite behaviours may be *terminated* or *aborted*. A terminated behaviour ends in a state where no guard is true. An aborted behaviour ends in a state in which termination is not guaranteed because there is an enabled action that itself may never terminate [BvW94].

Refinement is performed with respect to observable *traces* of the action system [BvW94]. Given a behaviour s we obtain its observable trace $tr(s)$ by

1. removing the local state component from each element, and
2. removing repeated elements from *finite* contiguous subsequences with the same global state component.

An *approximation* relation $\preceq$ is then defined for behaviours. A behaviour s approximates behaviour t, denoted $s \preceq t$, if

- s aborts and $tr(s)$ is a prefix of $tr(t)$, or
- neither s nor t abort and $tr(s) = tr(t)$.

We can then say that an abstract action system $\mathcal{A}$ is refined by a concrete action system $\mathcal{C}$, denoted $\mathcal{A} \sqsubseteq \mathcal{C}$, if every behaviour of $\mathcal{C}$ has an approximating behaviour in $\mathcal{A}$.

$$\mathcal{A} \sqsubseteq \mathcal{C} \stackrel{\text{def}}{=} \forall\, t \in beh(\mathcal{C}) \cdot \exists\, s \in beh(\mathcal{A}) \cdot s \preceq t$$

For defining real-time systems, however, action systems have significant limitations.

- Atomic, indivisible actions make it difficult to represent actions whose implementation will *consume* time.
- Interleaving semantics makes it difficult to model independent parallel actions that may *overlap* in global time.
- The model does not allow actions to be *pre-empted* at some arbitrary point during their execution. (Refinement rules exist that allow actions to be broken into smaller parts [Bac93], but the rules are complex, and they allow actions to be interrupted only at pre-determined points.)

3. Definition: timed action systems

This section defines a model for *timed action systems* that allows time-consuming, overlapping and pre-emptible actions to be specified.

For the purposes of this article we use a discrete time model (so that we can use Z integer operators for manipulating time, and because discrete time is assumed by scheduling theory models). Let absolute time $\mathbb{A}$, and durations of time $\mathbb{D}$, be represented as integers.

$$\mathbb{A} == \mathbb{Z}$$
$$\mathbb{D} == \mathbb{Z}$$

Syntactically a timed action system is identical to an untimed one, except that predicates may use a number of implicitly-declared auxiliary specification variables. Semantically a timed action system can be defined by augmenting a standard action system with auxiliary declarations and actions. In the most general case, an action system like $\mathcal{A}$ in Section 2 is defined in its 'timed' interpretation $\mathcal{T}$ to be extended as follows.

$$
\begin{array}{l}
\mathcal{T} = \mathbf{begin} \\
\quad \mathbf{var}\ \vec{x};\ up_{\vec{x}};\ ac_{\vec{x}};\ bs_{\vec{p}};\ ai_{\vec{p}} \bullet \\
\qquad A_0 \wedge T_0; \\
\qquad \mathbf{do} \\
\qquad [\,]_{i \in I}\ (\mathbf{g}A_i \wedge \mathbf{g}A_i^T) \rightarrow (\mathbf{s}A_i \wedge \mathbf{s}A_i^T) \\
\qquad [\,]\ \mathbf{g}T \rightarrow \mathbf{s}T \\
\qquad \mathbf{od} \\
\quad \mathbf{end} : \{\vec{z}, up_{\vec{z}}, ac_{\vec{z}}, now\}
\end{array}
$$

In these generic definitions, declarations that contain subscripted vectors are assumed to be duplicated for each vector element.

The auxiliary features are as follows. Not all are needed at every level of abstraction. Section 4 shows how they are added incrementally during system development.

- Variable *now*, of type $\mathbb{A}$, is declared. It is used to model the current global absolute time.
- Variables $up_{\vec{y}}$, of type $\mathbb{A}$, are declared for all local and global state variables $\vec{y}$. They are used to record the absolute time at which the corresponding state variable was most recently *updated* by an action.
- Variables $ac_{\vec{y}}$, of type $\mathbb{D}$, are declared for all local and global state variables $\vec{y}$. They are used to indicate that the corresponding state variable is being *accessed* by a time-consuming action for the given duration.
- Variables $bs_{\vec{p}}$, of type $\mathbb{D}$, are declared for all processes $\vec{p}$ from which the action system is constructed. They are used to indicate that the corresponding process is *busy* performing an action for the given duration.
- Variables $ai_{\vec{p}}$, of type $I \cup \{0\}$, are declared for all processes $\vec{p}$ from which the action system is constructed. They are used to *identify* which action, if any, the corresponding process is performing.
- An extra initialisation predicate T_0 is introduced. This is used to initialise the above auxiliary time variables.
- A new conjunct $\mathbf{g}A_i^T$ is added to the guard for each action A_i. This is used to ensure that the action can occur only when the auxiliary time variables have appropriate values.
- A new predicate $\mathbf{s}A_i^T$ is added to the statement part for each action A_i. This is used to update the auxiliary time variables whenever the action occurs.
- An auxiliary *tick* action T, with guard $\mathbf{g}T$ and statement part $\mathbf{s}T$, is added to the action system. This is used to model the effect of the passage of time on the auxiliary time variables.

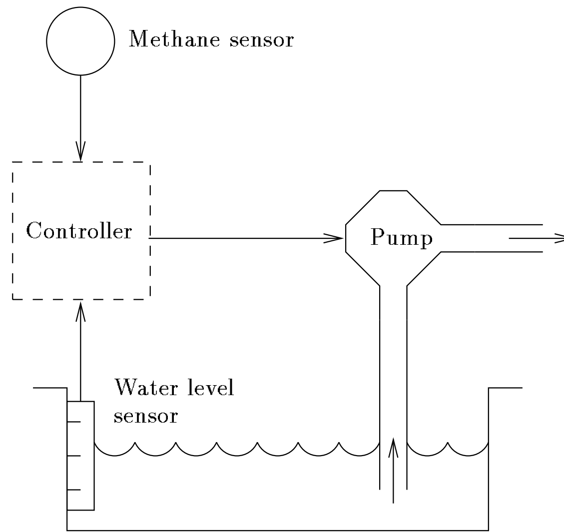

Fig. 1. Operating environment for the pump controller.

4. Development method

This section shows how the above auxiliary features can be systematically introduced and applied during the development of a real-time, multi-tasking system. A development procedure consisting of six major steps is outlined. Many smaller refinement steps may be applied at each level, of course, using the standard refinement rules established for action systems [BS91].

For each step a generic description is given and then illustrated with a concrete example based on the well-known mineshaft pump controller case study [BW90]. Figure 1 shows the environment in which the pump controller is intended to operate. The overall goal is to keep the depth of groundwater seeping into a mineshaft low enough to permit mining operations to continue unhindered. A sensor detects the depth of water and an electrically-powered pump is available for removing it. Unfortunately, the pump cannot be operated safely in atmospheres with high methane levels for fear of causing an explosion. (Nor can the pump be placed outside the mine due to the high pressures required. Water is often not pumped out of mines, but into disused shafts.) Therefore, the pump must be switched off if a methane sensor detects an unacceptably high gas concentration.

4.1. Timed traces

At the most abstract level, a real-time action system requirement is most succinctly expressed as the acceptable *timed traces* of the proposed system. A timed trace is, in turn, an abstraction of the *timed behaviours*.

A timed behaviour s is a sequence of states with a distinguished variable now denoting the absolute time at which this state was observable.

$$s = \langle (now_0, \vec{x}_0, \vec{z}_0), (now_1, \vec{x}_1, \vec{z}_1), \ldots \rangle$$

This variable allows the programmer to refer to the absolute time at which states should be observable. A valid timed behaviour is one where, in two successive states, the value of *now* is either unchanged or advances by exactly one time unit. Because the system may perform several independent state changes simultaneously, there is no requirement that *now* advances from one state to the next. Action system semantics represents parallelism by nondeterministic interleaving, so two or more *simultaneous* changes to disjoint parts of the state space appear in a timed behaviour as successive states with different state variable components but the same value of *now*.

Time does always advance, however, when we abstract away to timed traces. As in Section 2, this involves removing the local state component from each state (including our auxiliary time variables) and removing repetitive subsequences (where the value of *now* is also repeated). A new requirement, however, is to

3. remove all but the last element from finite contiguous subsequences with the same value of *now*.

In this way only an entire state change is reflected in a timed *trace*, even when the timed *behaviour* would reveal that the step was composed of several independent state changes occurring at the same moment.

Refinement of timed traces then follows directly from the untimed definition in Section 2, with *now* treated like any other variable.

4.1.1. Timed trace example

Let there be a scale, in millimetres, representing the depth of water in the mineshaft.

$$\mathrm{mm} == \mathbb{N}$$

There is a given maximum permissable depth for the water level.

$$\mid \ H_2Omax : \mathrm{mm}$$

Methane pressure can be measured using Pascals as a scale.

$$\mathrm{Pa} == \mathbb{N}$$

There is a given maximum methane limit for safe operation of mining equipment.

$$\mid \ CH_4max : \mathrm{Pa}$$

Our desired system is then specified with respect to the following three observable variables.

$$Time \mathrel{\widehat{=}} [\, now : \mathbb{A} \,]$$

$$Water \mathrel{\widehat{=}} [\, H_2O : \mathrm{mm} \,]$$

$$Methane \mathrel{\widehat{=}} [\, CH_4 : \mathrm{Pa} \,]$$

For expressing timed traces over some state S, first define those traces in which *now* appears in S, and advances by one time unit at each step, starting at time 0.

$$trace\ S == \{ s : \mathbb{N} \rightarrow S \mid \forall\, i : \mathbb{N} \bullet s(i).now = i \}$$

Our top-level requirement is then to create a system *Shaft* that obeys the following property.

$$tr(Shaft) = \{a : trace(Time \wedge Water \wedge Methane) \mid \forall t : \mathbb{A} \bullet (\forall u : 0 \,..\, t \bullet CH_4@u \leqslant CH_4max) \Rightarrow H_2O@t \leqslant H_2Omax\}$$

Here $v@t$ is a syntactic shorthand for $a(t).v$, denoting the value of variable v at time t [SZH93].

In other words, at all absolute times t, the water level H_2O must not exceed H_2Omax as long as, for all times u in the past, the methane level CH_4 did not exceed CH_4max. After the methane level passes this threshold, however, the system is no longer obliged to maintain low water levels. Of course, this does not say a methane level exceeding CH_4max must *cause* the mine to flood. In particular, methane peaks above CH_4max of very short duration may not even be detected and, even after mining equipment is forced to shut down, the seepage of water into the shaft may be so slow that its level remains low for some time.

This specification is quite abstract, merely relating the observable environmental variables to one another. In particular, it does not specify *how* low water levels are to be maintained, or any reaction times on equipment required to guarantee safety. This detail is introduced during the refinement below.

4.2. State machine

In the next major step, a state-machine solution to the traces requirement is developed. This is expressed as an *instantaneous* timed action system. This looks like a standard action system except that it may use auxiliary variables *now* and $up_{\vec{y}}$.

For each state variable v in $\vec{y}$, up_v denotes the last time v was *updated*. This allows the programmer to refer to the time at which the current value of v became observable, in the state-machine model. (Significantly, we can now see why trace-based specifications are more succinct than state-machine ones. Where a state-machine model requires auxiliary variables to keep track of past events, a trace-based specification has the full history of previous states available, without the need for additional variables.)

Semantically an instantaneous timed action system $\mathcal{A}$ can be defined as an augmented action system $\mathcal{T}$ as follows. (For brevity in the following generic Z definitions we take the liberty of letting operators such as Δ and Ξ precede variable names [PST91]. We also omit the types of auxiliary variables as these have already been given in Section 3. A predicate containing a vectored subscript denotes the conjunction of individual predicates for each vector element.)

- The auxiliary *now* and $up_{\vec{y}}$ variables are declared.
- The timed initialisation predicate asserts that the auxiliary variables are all initially zero.

T_0

$now;\ up_{\vec{y}}$

$now = 0 \wedge up_{\vec{y}} = 0$

- For each action A_i, that accesses variables $\vec{a}_i$, its guard $\mathbf{g}A_i$ may refer to the following implicitly-declared auxiliary time variables.

$$\boxed{\mathbf{g}A_i^T \quad now;\ up_{\vec{a}_i}}$$

- For each action A_i, that writes to variables $\vec{w}_i$ and reads variables $\vec{r}_i$, the statement $\mathbf{s}A_i$ is augmented so that all updated variables are marked as having been updated at time now.

$$\mathbf{s}A_i^T: \quad \Delta(up_{\vec{w}_i});\ \Xi(up_{\vec{r}_i}, now) \quad | \quad up'_{\vec{w}_i} = now$$

At this level of abstraction actions denote instantaneous state-changes of interest. They start and end at the same moment in time.

- The statement part of the auxiliary tick action increments now and leaves all other variables unchanged.

$$\mathbf{s}T: \quad \Delta(now);\ \Xi(\vec{y}, up_{\vec{y}}) \quad | \quad now' = now + 1$$

- The guard on the auxiliary tick action T is defined so that ticking will not cause any other true guard to become false.

$$\mathbf{g}T: \quad now;\ \vec{y};\ up_{\vec{y}} \quad | \quad \bigwedge_{i \in I} \left(\mathbf{g}A_i \Rightarrow \mathbf{g}A_i \left[\frac{now+1}{now}\right]\right)$$

Here $\mathbf{g}A_i[now+1/now]$ denotes guard $\mathbf{g}A_i$ with 'now' replaced by '$now+1$'. This guard ensures that a tick occurs only as long it does not disable any currently enabled guard $\mathbf{g}A_i$. (A tick may enable a disabled guard, however.) If some guard $\mathbf{g}A_i$ is true, and performing a tick would not cause any guard to become false, then there is a nondeterministic choice between doing action A_i or performing a tick. In effect, this tick guard allows each action A_i to occur at *any* of the times specified by $\mathbf{g}A_i$, without the danger that the passage of time will cause an action to miss a deadline [AL94].

For such an augmented action system to be a valid refinement of a timed trace specification, the set of its traces must be a subset of those defined by the trace specification. Here we have introduced new auxilary time variables $up_{\vec{y}}$. This is a straightforward application of *superposition refinement* in action systems [Bac92], a special case of data refinement in which new variables are added only.

4.2.1. State machine example

We now introduce an action system whose traces satisfy the requirement expressed in Section 4.1.1. In doing so we also introduce aspects of the anticipated physical environment and the anticipated solution.

Let there be a known maximum possible change in the methane level in one time unit.

$$\delta CH_4 : \mathrm{Pa}$$

Also define a period CH_4per within which the methane level must be sampled at least once, with a small possible variation of no more than CH_4err. (These safety-critical durations may be laid down by legislation.)

$$CH_4per, CH_4err : \mathbb{D}$$

$$CH_4per > CH_4err$$

Note that if the methane level briefly rises above CH_4max for a time significantly less than CH_4per then this short peak may not be detected. However if there is a period as long as $CH_4per + CH_4err$ during which the methane level is *continuously* high then it *must* be detected.

Naturally we expect water levels to be managed by some sort of pump. Let δH_2Oin define the maximum possible rate of water seepage into the mine, and δH_2Oout be the maximum possible rate at which water can be pumped out of the shaft against a worst-case inflow, both expressed as the change of water level in one time unit.

$$\delta H_2Oin, \delta H_2Oout : \mathrm{mm}$$

$$\delta H_2Oin \leqslant \delta H_2Oout$$

To make a solution feasible we require that the pump can remove water at least as fast as it seeps into the mine.

In practice any implementation will take some to time react to rising water levels approaching H_2Omax and get the pump operating. If the water level is never to exceed H_2Omax then we must start work towards switching the pump on *before* this level is reached. Let H_2Omark denote the water level at which the system must react to rising water, and relate this to duration *react* as follows [MH92].

$$react : \mathbb{D};\ H_2Omark : \mathrm{mm}$$

$$H_2Omark < H_2Omax$$

$$react < \left\lceil \frac{H_2Omax - H_2Omark}{\delta H_2Oin} \right\rceil$$

$$react < CH_4per - CH_4err$$

We also use *react* to denote the time required to switch the pump *off*, so the last predicate relates it to the sampling period of methane levels in order to place an upper bound on how quickly the pump will be stopped when dangerous methane levels are detected [LB90].

Some local variables are needed to control the action system. These are the status of the water pump, a signal that the water level exceeds H_2Omark, an indication that a dangerous level of methane has been detected, and a counter

used to keep track of methane sensing periods.

$$
\begin{aligned}
Pump &\mathrel{\widehat=} [\, pumping : \mathbb{B} \,] \\
Depth &\mathrel{\widehat=} [\, deep : \mathbb{B} \,] \\
Danger &\mathrel{\widehat=} [\, shutdown : \mathbb{B} \,] \\
Counter &\mathrel{\widehat=} [\, n : \mathbb{N} \,]
\end{aligned}
$$

The action system specification is then expressed in terms of six distinct timed actions.

$$
\begin{array}{l}
Shaft = \mathbf{begin} \\
\quad \mathbf{var}\ pumping;\ deep;\ shutdown;\ n \bullet \\
\quad\quad Init; \\
\quad\quad \mathbf{do} \\
\quad\quad\quad WaterLevel \\
\quad\quad [\!]\ MethaneLevel \\
\quad\quad [\!]\ CheckWater \\
\quad\quad [\!]\ CheckMethane \\
\quad\quad [\!]\ PumpOff \\
\quad\quad [\!]\ PumpOn \\
\quad\quad \mathbf{od} \\
\quad \mathbf{end} : \{H_2O, CH_4\}
\end{array}
$$

Of course many other features, including the tick action and declarations of auxiliary variables, are assumed to exist implicitly, as per the definitions above.

Initially the pump is off, the period counter for methane monitoring is zero, no shutdown or deep water conditions have been diagnosed, and we assume the system is started only when the water and methane levels are low.

$$
\begin{array}{|l}
\hline
\textit{Init} \\
Pump;\ Danger;\ Depth;\ Counter;\ Water;\ Methane \\
\hline
\neg\ pumping \wedge \neg\ shutdown \wedge \neg\ deep \wedge n = 0 \\
H_2O < H_2Omark \\
CH_4 \leqslant CH_4max \\
\hline
\end{array}
$$

The first two actions model properties of the physical environment. Action *WaterLevel* defines the change in the water depth in each time unit. Its guard ensures that it is performed once at every unit of time.

$$
\begin{array}{|l}
\hline
\mathbf{g}\ WaterLevel \\
Water \\
\hline
now = up_{H_2O} + 1 \\
\hline
\end{array}
$$

As soon as *now* advances past the last update time for variable H_2O the guard is enabled and action *WaterLevel* must occur before *now* can advance further. (The guard thus places an upper bound on the time at which each instance of *WaterLevel* must occur. Merely stating $now > up_{H_2O}$ would be too weak because it allows *now* to advance indefinitely before performing the action. The passage of time would never disable such a guard.) The action itself changes the water level nondeterministically. If the pump is off the water level may increase by up to δH_2Oin units. If the pump is on the water level may decrease by up to

δH_2Oout units.

s *WaterLevel*

$$\Xi Pump;\ \Delta Water$$

$$pumping \Rightarrow H_2O' \in (H_2O - \delta H_2Oout) \mathrel{..} H_2O$$
$$\neg\ pumping \Rightarrow H_2O' \in H_2O \mathrel{..} (H_2O + \delta H_2Oin)$$

Since the action updates H_2O, thus implicitly setting up_{H_2O} equal to now, the action is then disabled until time advances again. (Not all variables are declared in the above schema. By action system convention variables not mentioned in an action are left unchanged.)

Similarly, action *MethaneLevel* occurs once every time unit.

g *MethaneLevel*

$$Methane$$

$$now = up_{CH_4} + 1$$

It allows the level of methane in the shaft to vary randomly by up to δCH_4 units.

s *MethaneLevel*

$$\Delta Methane$$

$$CH_4{}' \in CH_4 \pm \delta CH_4$$

Action *CheckWater* maintains the *deep* variable which is true only while the water level equals or exceeds H_2Omark. The guard ensures the action occurs whenever the value of *deep* does not agree with the actual depth of the water, immediately after each update to H_2O.

g *CheckWater*

$$Water;\ Depth$$

$$deep \Leftrightarrow H_2O < H_2Omark$$
$$up_{H_2O} = now$$

The action thus reacts to changes in the water level instantly, and corrects the value of *deep* accordingly.

s *CheckWater*

$$\Xi Water;\ \Delta Depth$$

$$deep' \Leftrightarrow H_2O \geqslant H_2Omark$$

Action *CheckMethane* represents the act of periodically checking the methane level and updating the *shutdown* variable. The guard ensures that the action is performed every CH_4per time units, within a deadline of CH_4err.

g *CheckMethane*

$$Counter$$

$$now \in (n * CH_4per) \mathrel{..} (n * CH_4per + CH_4err)$$

The action body increments the period counter and makes *shutdown* true only

if the methane level is too high.

$$
\begin{array}{l}
\mathbf{s}\,CheckMethane \\
\hline
\Xi Methane;\ \Delta Danger;\ \Delta Counter \\
\hline
n' = n + 1 \\
shutdown' \Leftrightarrow CH_4 > CH_4max
\end{array}
$$

The last two actions switch the pump off and on according to current conditions in the mine shaft. *PumpOff* switches the pump off either when the water level is no longer too deep or when a shutdown has been signalled.

$$
\begin{array}{l}
\mathbf{g}\,PumpOff \\
\hline
Danger;\ Depth;\ Pump \\
\hline
pumping \\
\neg\ deep \lor shutdown \\
\neg\ deep \Rightarrow now \leqslant up_{deep} + react \\
shutdown \Rightarrow now \leqslant up_{shutdown} + react
\end{array}
$$

This must be done within *react* time units of either of these conditions becoming true, as indicated by the last two conjuncts in the guard. This particular form of timing requirement ensures that if both conditions $\neg\ deep$ and *shutdown* become true within a short time of one another then *PumpOff* will be performed within *react* time units of the *earlier* event. The action itself merely requires the pump to be switched off.

$$
\begin{array}{l}
\mathbf{s}\,PumpOff \\
\hline
\Delta Pump \\
\hline
\neg\ pumping'
\end{array}
$$

(Readers may wonder why we bother to switch the pump off at all. Leaving it on permanently, as long as the methane levels are not too high, is guaranteed to satisfy the specification in Section 4.1.1! However, with reference to the complete example [BW90], we note that the pump should not be allowed to run 'dry' for fear of damaging itself and, more seriously, operation of the pump generates carbon monoxide, with obvious safety concerns for the miners.)

Action *PumpOn* can occur if the water is too deep and the methane level is low.

$$
\begin{array}{l}
\mathbf{g}\,PumpOn \\
\hline
Danger;\ Depth;\ Pump \\
\hline
\neg\ pumping \\
deep \land \neg\ shutdown \\
now \leqslant max\{up_{deep}, up_{shutdown}\} + react
\end{array}
$$

Since we must wait for both of these conditions to be true, the timing requirement states that the pump must respond within *react* time units of the *later* event. (Our state-machine model thus allows the pump to be switched on again after a high methane concentration subsides, although this feature was not *required* by top-level specification *Shaft*.) Again the statement part is trivial.

s$PumpOn$

$\Delta Pump$
$pumping'$

Proof that this state machine model satisfies the specification given in Section 4.1.1 follows by noting the possible sequence of changes to the observable variables. The required trace behaviour is violated if the water level exceeds H_2Omax when the methane levels have never been above CH_4max. Action *WaterLevel* is the only one that can increase the depth of the water, and it does so only if *pumping* is false. Assuming that the water level is rising, there will be a moment when H_2O reaches or exceeds H_2Omark. At that time action *CheckWater* will occur and raise the *deep* signal, and within *react* time units of this happening *PumpOn* must occur, which prevents *WaterLevel* from increasing the depth any further until either the water level goes below H_2Omark or high methane levels force the pump to be switched off.

4.3. Time-consuming actions

At the next level of abstraction we introduce the notion that those actions to be implemented as executable code cannot be performed instantaneously.

For each state variable v in $\vec{y}$, ac_v marks the duration for which v is not usable because it is already being *accessed* by a time-consuming action. These periods of inaccessiblilty are used to model the notion that actions take time to update variables.

For each action A_i, the programmer must supply an expression $\mathbf{e}A_i$, of type $\mathbb{P}\mathbb{D}$, which returns the set of acceptable (non-negative) *execution times* for this action. Expression $\mathbf{e}A_i$ may refer to any variables accessed by action A_i, including the time variables. In practice, $\mathbf{e}A_i$ embodies estimates of the best and worst-case execution times for the ultimate implementation of A_i. Typically the initial values will be pessimistic. Later refinements can tighten the estimates, as more implementation detail becomes available.

Semantically a *time-consuming* timed action system is defined as a further extension to the augmented action system of Section 4.2 as follows.

- Auxiliary $ac_{\vec{y}}$ variables are declared.
- The initialisation predicate is extended to assert that all the $ac_{\vec{y}}$ variables are initially zero.

T_0

$now;\ up_{\vec{y}};\ ac_{\vec{y}}$
$now = 0 \wedge up_{\vec{y}} = 0 \wedge ac_{\vec{y}} = 0$

- Each guard $\mathbf{g}A_i$ is extended so that action A_i cannot begin execution while any variable that it wants to use is currently being accessed, i.e., if any $ac_{\vec{a}_i}$ is greater than zero.

$$\begin{array}{|l}\hline \mathbf{g}A_i^T \\ now;\ up_{\vec{a}_i};\ ac_{\vec{a}_i} \\ \hline ac_{\vec{a}_i} \leqslant 0 \\ \hline\end{array}$$

- Each statement $\mathbf{s}A_i$ is extended so that all variables $\vec{a}_i$ it uses are marked as inaccessible for the duration e in $\mathbf{e}A_i$ that the atomic action is 'executing'. Also the update times for variables $\vec{w}_i$ that are updated by this action are set to the *future* absolute time at which the time-consuming action nominally finishes.

$$\begin{array}{|l}\hline \mathbf{s}A_i^T \\ \Delta(\vec{a}_i, up_{\vec{w}_i}, ac_{\vec{a}_i});\ \Xi(now, up_{\vec{r}_i}) \\ \hline \exists\, e : \mathbf{e}A_i \bullet \\ \qquad ac'_{\vec{a}_i} = e \wedge up'_{\vec{w}_i} = now + e \\ \hline\end{array}$$

(The accessed state variables $\vec{a}_i$, and the absolute time now, are declared here in case they are referred to in $\mathbf{e}A_i$.)

- The guard on the tick action is extended so that it takes account of the extra condition on time variables added to the other guards above.

$$\begin{array}{|l}\hline \mathbf{g}T \\ now;\ \vec{y};\ up_{\vec{y}};\ ac_{\vec{y}} \\ \hline \bigwedge_{i \in I} \left((\mathbf{g}A_i \wedge \mathbf{g}A_i^T) \Rightarrow (\mathbf{g}A_i \wedge \mathbf{g}A_i^T) \left[\frac{now + 1}{now} \right] \right) \\ \hline\end{array}$$

- The tick action is extended so as to decrement the duration for which each variable is inaccessible, whenever time passes.

$$\begin{array}{|l}\hline \mathbf{s}T \\ \Delta(now, ac_{\vec{y}});\ \Xi(\vec{y}, up_{\vec{y}}) \\ \hline now' = now + 1 \\ ac'_{\vec{y}} = ac_{\vec{y}} - 1 \\ \hline\end{array}$$

Each time a tick occurs all access counters are decremented, possibly 'releasing' inaccessible variables.

Thus each action still occurs 'instantaneously' (at absolute time now) but its effect on a variable v cannot be observed by any other action until ac_v becomes zero (at time $now + e$). Similarly, although a variable v is updated by an action as soon as it begins (at time now), the corresponding up_v variable is set to indicate that the update is observable only at the nominal end of the action (at time $now + e$). It is this approach that allows us to model time-consuming actions, that may overlap in time, in an interleaving framework. Since actions access variables under mutual exclusion, no other action can 'see' when a variable being updated is actually modified, so no anomalies are introduced by this model.

For a time-consuming action to be a valid refinement of an instantaneous one, it must *finish* creating its new state in the range of times at which the corresponding instantaneous action occurred. This means that the range of possible starting times expressed in the guard must be made earlier to allow for the amount of

time consumed by the action statement. Also, since the time-consuming action accesses variables at the time it *starts*, then these initial variable values must correspond to past values referred to by the equivalent instantaneous action. Furthermore, the periods for which each variable is designated 'inaccessible' must correspond to times at which only one action made use of this variable in the instantaneous system.

4.3.1. *Time-consuming actions example*

For each action in the mineshaft design we must supply an expression defining its possible execution times. In this example we show this by following the name of an action statement predicate with the expression.

For instance, define durations denoting how much time it takes to perform action *CheckMethane* in the best and worst possible cases, respectively.

$$\begin{array}{|l} CMmin, CMmax : \mathbb{D} \\ \hline CMmin \leqslant CMmax \\ CMmax < CH_4err \end{array}$$

Then '$\mathbf{s}CheckMethane$ **in** $CMmin \mathinner{..} CMmax$' denotes the statement part of action *CheckMethane* augmented as described above, i.e., where $\mathbf{e}CheckMethane = CMmin \mathinner{..} CMmax$.

Modifying *CheckMethane* in this way creates a problem, however, because it shares variable CH_4 with action *MethaneLevel*. The time-consuming version of *CheckMethane* may now lock this variable for up to *CMmax* time units, thus making it impossible to perform action *MethaneLevel* at every tick, as required. This is not surprising—the instantaneous model gives each action the power to examine the value of any variable, at any time, without blocking other actions. When actions consume time this ability is lost. Further implementation detail must be introduced to overcome this.

We can prevent *CheckMethane* from blocking the *MethaneLevel* action by interposing another action between them so that they no longer share variables. Rather than assuming that *CheckMethane* can *directly* access the CH_4 variable, which represents the actual methane pressure in the environment, we introduce a new variable to model the pressure as seen by the methane sensing hardware.

$$MethaneGauge \mathrel{\widehat=} [\, CH_4gauge : \mathrm{Pa} \,]$$

We then replace *CheckMethane* with a time-consuming version that reads from this variable, rather than directly from CH_4. Since *CheckMethane* now takes up to *CMmax* time units to execute, its range of starting times is made correspondingly earlier.

$$\begin{array}{|l} \hline \mathbf{g}CheckMethane \\ Counter \\ \hline now \in (n * CH_4per) \mathinner{..} (n * CH_4per + CH_4err - CMmax) \\ \hline \end{array}$$

This guarantees that the time-consuming statement still finishes before CH_4err time units elapse, even in the worst situation. The modified statement can now refer to CH_4gauge instead of CH_4.

$$\begin{array}{l}
\mathbf{s}\mathit{CheckMethane}\ \mathbf{in}\ \mathit{CMmin} \mathinner{\ldotp\ldotp} \mathit{CMmax} \\
\hline
\Xi \mathit{MethaneGauge};\ \Delta \mathit{Danger};\ \Delta \mathit{Counter} \\
\hline
n' = n + 1 \\
\mathit{shutdown}' \Leftrightarrow \mathit{CH_4gauge} > \mathit{CH_4max}
\end{array}$$

We must next ensure that the reading given by CH_4gauge accurately reflects the true value of CH_4. This is done by adding a new action *SenseMethane* to update CH_4gauge whenever CH_4 changes.

$$\begin{array}{l}
\mathbf{g}\mathit{SenseMethane} \\
\hline
\mathit{Methane};\ \mathit{MethaneGauge} \\
\hline
\mathit{now} = \mathit{up}_{\mathit{CH_4}} \\
\mathit{up}_{\mathit{CH_4gauge}} < \mathit{up}_{\mathit{CH_4}}
\end{array}$$

$$\begin{array}{l}
\mathbf{s}\mathit{SenseMethane}\ \mathbf{in}\ \{0\} \\
\hline
\Xi \mathit{Methane};\ \Delta \mathit{MethaneGauge} \\
\hline
\mathit{CH_4gauge}' = \mathit{CH_4}
\end{array}$$

In effect, this action represents the interface between the methane sensor and our computer system. Anticipating that this action will be implemented by hardware, with a negligible delay, we have specified that it occurs in 'zero' time (i.e., quicker than our discrete-time granularity). It is action *SenseMethane* that may now be blocked while *CheckMethane* executes, rather than *MethaneLevel*.

Since the *MethaneLevel* action itself models a continuous *physical* process it is also considered to occur instantaneously in our discrete-time model. Similarly for the *WaterLevel* action.

$$\begin{array}{ll}
\mathbf{e}\mathit{MethaneLevel} & = \{0\} \\
\mathbf{e}\mathit{WaterLevel} & = \{0\}
\end{array}$$

Accounting for the time required to perform the *CheckWater* action introduces a problem like that found with *CheckMethane* above and we adopt a similar solution. Let there be a new local variable representing the computer system's view of the actual water level.

$$\mathit{WaterGauge} \mathrel{\widehat{=}} [\ \mathit{H_2Ogauge} : \mathrm{mm}\ ]$$

We also introduce a duration denoting the acceptable lag between significant changes in the water level and the time they are noted by the system. Ideally this should be quite small, and *must* be less than the overall reaction time of the system to such changes.

$$\begin{array}{|l}
\mathit{H_2Olag} : \mathbb{D} \\
\hline
\mathit{H_2Olag} < \mathit{react}
\end{array}$$

As with the methane gauge we introduce a new action *SenseWater* to model the (effectively instantaneous) hardware connection between the water sensor attached to the mineshaft wall and the water gauge input variable.

$\mathbf{g}\mathit{SenseWater}$
$\mathit{Water};\ \mathit{WaterGauge}$

$now = up_{H_2O}$
$up_{H_2Ogauge} < up_{H_2O}$

$\mathbf{s}\mathit{SenseWater}\ \mathbf{in}\ \{0\}$
$\Xi\,\mathit{Water};\ \Delta\,\mathit{WaterGauge}$

$H_2Ogauge' = H_2O$

We then replace action *CheckWater* with a new version which is still triggered at the same time, but can take up to H_2Olag time units to execute.

$\mathbf{g}\mathit{CheckWater}$
$\mathit{WaterGauge};\ \mathit{Depth}$

$deep \Leftrightarrow H_2Ogauge < H_2Omark$
$up_{H_2Ogauge} = now$

$\mathbf{s}\mathit{CheckWater}\ \mathbf{in}\ 0\,..\,H_2Olag$
$\Xi\,\mathit{WaterGauge};\ \Delta\,\mathit{Depth}$

$deep' \Leftrightarrow H_2Ogauge \geqslant H_2Omark$

This execution-time delay means that the actual water level may have risen by up to $H_2Olag * \delta H_2Oin$ millimetres, or fallen by up to $H_2Olag * \delta H_2Oout$ millimetres, before the change is observed by the system! Since this potential delay affects our ability to respond in a timely fashion to changes in water level, it must be accounted for in the revision of the pump actions below.

Obviously the pump cannot be switched on and off instantaneously, and again we are faced with the problem that making *PumpOff* and *PumpOn* consume time will block the *WaterLevel* action due to the shared variable *pumping*. Again the solution is to add further implementation detail. Whereas the *pumping* variable used in *WaterLevel* represents the actual state of the physical pump, we introduce a new variable to denote the status of the pump as seen by our system.

$$PumpStatus \mathrel{\widehat{=}} [\,pumpon : \mathbb{B}\,]$$

Let the following constants be the minimum and maximum times required to decide to switch the pump on or off.

$POmin, POmax : \mathbb{D}$

$POmin \leqslant POmax$
$POmax < react - H_2Olag$

Notice that the available reaction time has been reduced by the worst-case lag in sensing water level changes. Action *PumpOff* is then replaced as follows.

$$\begin{array}{|l}
\textbf{g}\mathit{PumpOff} \\
\hline
\mathit{Danger};\ \mathit{Depth};\ \mathit{PumpStatus} \\
\hline
\mathit{pumpon} \\
\neg\ \mathit{deep} \lor \mathit{shutdown} \\
\neg\ \mathit{deep} \Rightarrow \mathit{now} \leqslant \mathit{up}_{\mathit{deep}} + \mathit{react} - \mathit{H_2Olag} - \mathit{POmax} \\
\mathit{shutdown} \Rightarrow \mathit{now} \leqslant \mathit{up}_{\mathit{shutdown}} + \mathit{react} - \mathit{POmax} \\
\hline
\end{array}$$

$$\begin{array}{|l}
\textbf{s}\mathit{PumpOff}\ \textbf{in}\ \mathit{POmin} \mathinner{..} \mathit{POMax} \\
\hline
\Xi \mathit{Danger};\ \Xi \mathit{Depth};\ \Delta \mathit{PumpStatus} \\
\hline
\neg\ \mathit{pumpon}' \\
\hline
\end{array}$$

Again the guard has been changed to account for the execution time of the action. Further, the potential lag in reading water levels reduces the available reaction time when a change to *deep* triggers the action. A corresponding change for *shutdown* is unnecessary because a potential delay in reading methane levels is accounted for by $CH_4 err$.

A similar modification is made to *PumpOn*.

$$\begin{array}{|l}
\textbf{g}\mathit{PumpOn} \\
\hline
\mathit{Danger};\ \mathit{Depth};\ \mathit{PumpStatus} \\
\hline
\neg\ \mathit{pumpon} \land \mathit{deep} \land \neg\ \mathit{shutdown} \\
\mathit{now} \leqslant \mathit{max}\{(\mathit{up}_{\mathit{deep}} - \mathit{H_2Olag}), \mathit{up}_{\mathit{shutdown}}\} + \mathit{react} - \mathit{POmax} \\
\hline
\end{array}$$

$$\begin{array}{|l}
\textbf{s}\mathit{PumpOn}\ \textbf{in}\ \mathit{POmin} \mathinner{..} \mathit{POmax} \\
\hline
\Xi \mathit{Danger};\ \Xi \mathit{Depth};\ \Delta \mathit{PumpStatus} \\
\hline
\mathit{pumpon}' \\
\hline
\end{array}$$

Another new 'blockable' action *SignalPump* is introduced to keep *pumpon* up to date with *pumping*, in effect modelling the physical wiring between the pump and the computer.

$$\begin{array}{|l}
\textbf{g}\mathit{SignalPump} \\
\hline
\mathit{PumpStatus};\ \mathit{Pump} \\
\hline
\mathit{now} = \mathit{up}_{\mathit{pumpon}} \\
\mathit{up}_{\mathit{pumping}} < \mathit{up}_{\mathit{pumpon}} \\
\hline
\end{array}$$

$$\begin{array}{|l}
\textbf{s}\mathit{SignalPump}\ \textbf{in}\ \{0\} \\
\hline
\Xi \mathit{PumpStatus};\ \Delta \mathit{Pump} \\
\hline
\mathit{pumping}' = \mathit{pumpon} \\
\hline
\end{array}$$

In refinement terms this revised system design still represents a valid development of its predecessor in Section 4.2.1. The new state variables are local only and hence not externally observable. The observable CH_4 variable is still updated only by action *MethaneLevel* which is unchanged. The observable H_2O variable is influenced by changes to local variable *pumping* and, in turn, actions

PumpOff and *PumpOn*. The changes to their guards ensure that these actions still complete within *react* time units of significant events in the environment, as required. The new actions *SenseMethane*, *SenseWater* and *SignalPump* all serve to copy local variables only.

This system can generate fewer traces than the instantaneous version, however, because action *MethaneGauge* can now never complete in *less* than *CMmin* time units from the start of each CH_4per period. Similarly, the pump actions will never react more quickly than *POmin* time units to changes in the environment. The time-consuming system is thus more deterministic than the instantaneous one in Section 4.2.1.

4.4. Processes

In the shared variable approach for action systems the programmer defines named, disjoint sets of actions to represent *processes* [Bac92]. For example, we state that process q is formed from actions A_j to A_m as follows.

$$q = \{A_j, \ldots, A_m\}$$

Let $\vec{p}$ denote the set of such processes defined for the action system of interest.

For each process q in $\vec{p}$, bs_q denotes those times at which q is *busy* performing a time-consuming action and therefore cannot start another action. These variables are used to ensure that each process can perform only one action at a time.

Semantically a *process-based* timed action system is defined through further extensions to the model presented in Section 4.3.

- Auxiliary $bs_{\vec{p}}$ variables are declared.
- The initialisation predicate is extended to assert that all the busy variables are initially zero.

$$\begin{array}{|l} \hline T_0 \\ now;\ up_{\vec{y}};\ ac_{\vec{y}};\ bs_{\vec{p}} \\ \hline now = 0 \wedge up_{\vec{y}} = 0 \wedge ac_{\vec{y}} = 0 \wedge bs_{\vec{p}} = 0 \\ \hline \end{array}$$

- Each guard $\mathbf{g}A_i$, for some action belonging to process q, is extended so that action A_i cannot start while some preceding action in process q is still busy.

$$\begin{array}{|l} \hline \mathbf{g}A_i^T \\ now;\ up_{\vec{a}_i};\ ac_{\vec{a}_i};\ bs_q \\ \hline ac_{\vec{a}_i} \leqslant 0 \wedge bs_q \leqslant 0 \\ \hline \end{array}$$

- Statement $\mathbf{s}A_i$, for some action belonging to process q, is extended so that the process is marked as busy for the execution time e in $\mathbf{e}A_i$ of the action.

$$\begin{array}{|l} \hline \mathbf{s}A_i^T \\ \Delta(\vec{a}_i, up_{\vec{w}_i}, ac_{\vec{a}_i}, bs_q);\ \Xi(now, up_{\vec{r}_i}) \\ \hline \exists\, e : \mathbf{e}A_i \bullet \\ \quad ac'_{\vec{a}_i} = e \wedge up'_{\vec{w}_i} = now + e \wedge bs'_q = e \\ \hline \end{array}$$

- The statement part of the tick action is redefined so that each time a tick occurs all busy counters $bs_{\vec{p}}$ are decremented, modelling the uniform passage of global time across all processes.

$$
\begin{array}{|l}
\hline
\mathbf{s}\,T \\
\Delta(now, ac_{\vec{y}}, bs_{\vec{p}});\ \Xi(\vec{y}, up_{\vec{y}}) \\
\hline
now' = now + 1 \\
ac'_{\vec{y}} = ac_{\vec{y}} - 1 \\
bs'_{\vec{p}} = bs_{\vec{p}} - 1 \\
\hline
\end{array}
$$

Thus actions in the same process are made atomic with respect to one another. In other words, each process must perform the actions from which it is composed one at a time. True parallelism (modelled using interleaving!) is still assumed *between* processes, however.

For a process-based action system to be a valid refinement of a time-consuming one it must be shown that the reduction in available concurrency, due to actions grouped into the same process being mutually exclusive, still allows valid behaviours.

4.4.1. Processes example

We now formalise the allocation of mineshaft actions to logical processes. In general the entire system can be partitioned into the physical environment, and the computer system to be implemented.

$$Shaft = Environment \cup System$$

The physical environment model consists of two actions, modelling the external properties of interest.

$$Environment = \{WaterLevel, MethaneLevel\}$$

The computer system can be divided into its hardware and software components.

$$System = Hardware \cup Software$$

The hardware actions consist of sensing the water and methane levels and sending actuator signals to the pump.

$$Hardware = \{SenseMethane, SenseWater, SignalPump\}$$

Finally, the software component has three distinct repetitive behaviours to perform.

$$
\begin{aligned}
Software &= MonitorWater \cup MonitorMethane \cup ControlPump \\
MonitorWater &= \{CheckWater\} \\
MonitorMethane &= \{CheckMethane\} \\
ControlPump &= \{PumpOff, PumpOn\}
\end{aligned}
$$

There is no need to alter the actions themselves further in this example. The only actions that may be in contention due to this partitioning are instantaneous

ones, such as *WaterLevel* and *MethaneLevel*, which cannot inhibit one another's progress, and *PumpOff* and *PumpOn*, which are already mutually exclusive.

This partitioning determines which variables are shared between processes. In particular note that the guards on actions *PumpOff* and *PumpOn* involve variables that are shared with both *Hardware* and the software processes *MonitorWater* and *MonitorMethane*.

4.5. Execution environment

The next level of abstraction introduces physical *processors* and their *operating systems* to the model. The programmer allocates each process to a processor. For instance, we state that processes q to r are allocated to processor C as follows.

$$C = \{q, \ldots, r\}$$

We give definitions below for a group of processes $\vec{p}$ all allocated to the same processor.

Features of the target operating system environment are captured at this level. Below we assume that a *static-priority pre-emptive scheduling policy* is anticipated [ABR+93]. The programmer specifies a *priority ordering* $\mathcal{P}$ between the actions. This is represented as a partial, transitive ordering. This may be defined as follows, where action A_j has a higher priority than action A_k, and so on.

$$\mathcal{P} = \{A_j \; A_k, \ldots\}$$

Ideally this ordering should obey scheduling theory principles [ABR+93]. All actions belonging to the same process, that access variables local to that process only, should have the same *base* priority. We also expect that base priorities are unique to each process. To avoid the danger of priority inversion, priority inheritance principles should be respected [SRL90], using a shared variable access protocol like the *ceiling locking protocol* [II94]. Any action that accesses a variable shared with another process must be allocated a *ceiling* priority at least as high as the base priority of *any* process which may access that variable. Actions belonging to processes that reside on different processors should be left unordered—it is not meaningful to define a priority ordering across processor boundaries [Pil91].

For each process q in $\vec{p}$, there is a variable ai_q, identifying which action from I that process is currently performing, or 0 if the process is currently idle. This allows the scheduling model to determine the current *active* priority of the process. Also we can use action identifier ai_q to determine which variables process q has locked, i.e., $\vec{a}_{ai_q}$. (The $ac_{\vec{y}}$ variables tell us which variables in $\vec{y}$ are locked, but not by whom.)

Semantically a *scheduled* timed action system is defined as a further extension of the model from Section 4.4.

- Auxiliary $ai_{\vec{p}}$ variables are declared.
- The initialisation predicate is extended to assert that all $ai_{\vec{p}}$ variables are initially the null action.

$$
\begin{array}{|l}
\hline
T_0 \\
now;\ up_{\vec{y}};\ ac_{\vec{y}};\ bs_{\vec{p}};\ ai_{\vec{p}} \\
\hline
now = 0 \land up_{\vec{y}} = 0 \land ac_{\vec{y}} = 0 \land bs_{\vec{p}} = 0 \land ai_{\vec{p}} = 0 \\
\hline
\end{array}
$$

- Each guard $\mathbf{g}A_i$, for some action in process q residing on processor C, is extended so that A_i cannot start if there is *any* higher-priority action A_h on processor C that is also ready to begin.

$$
\begin{array}{|l}
\hline
\mathbf{g}A_i^T \\
now;\ up_{\vec{y}};\ ac_{\vec{y}};\ bs_{\vec{p}};\ ai_{\vec{p}} \\
\hline
ac_{\vec{a}_i} \leqslant 0 \\
bs_q \leqslant 0 \\
\mathbf{let}\ H == \{h : I \mid (A_h \mapsto A_i) \in \mathcal{P}^+\} \bullet \\
\qquad \neg \bigvee_{h \in H} (\mathbf{g}A_h \land \mathbf{g}A_h^T) \\
\hline
\end{array}
$$

Index set H identifies all actions A_h on processor C that have a higher priority than A_i.

- Statement $\mathbf{s}A_i$, for some action in process q residing on processor C, is extended so that the action identifier is recorded in ai_q.

$$
\begin{array}{|l}
\hline
\mathbf{s}A_i^T \\
\Delta(\vec{a}_i, up_{\vec{w}_i}, ai_q, ac_{\vec{a}_i}, bs_q);\ \Xi(now, up_{\vec{r}_i}) \\
\hline
\exists\, e : \mathbf{e}A_i \bullet \\
\qquad ac'_{\vec{a}_i} = e \land up'_{\vec{w}_i} = \infty \land bs'_q = e \land ai'_q = i \\
\hline
\end{array}
$$

The $up_{\vec{w}_i}$ variables are here set to the 'yet-to-be-defined' value ∞ because potential pre-emption makes it impossible to predict exactly when action A_i will finish in *absolute* time at the moment when the action begins. These variables are set to the actual finish time by the tick action below, when the action is known to have completed.

- The guard on the tick action is changed so that a tick cannot occur if there is any other action A_i that is ready. Thus each action A_i will begin at the *earliest* possible time.

$$
\begin{array}{|l}
\hline
\mathbf{g}T \\
now;\ \vec{y};\ up_{\vec{y}};\ ac_{\vec{y}};\ bs_{\vec{p}};\ ai_{\vec{p}} \\
\hline
\neg \bigvee_{i \in I} (\mathbf{g}A_i \land \mathbf{g}A_i^T) \\
\hline
\end{array}
$$

Whereas our previous specifications have allowed each action to start at any of its specified *release* times, modelling the actual behaviour of a scheduler requires us to capture the fact that the processor will never idle while there is an action that can be performed.

- The tick action is redefined so that it maintains the auxiliary time variables in a way that models a real-time scheduler's behaviour.

s T

$$\Delta(now, ac_{\vec{y}}, bs_{\vec{p}}, up_{\vec{y}}, ai_{\vec{p}});\ \Xi(\vec{y})$$

$$now' = now + 1$$

$$bs'_{\vec{p}} = \begin{cases} bs_{\vec{p}} - 1, & bs_{\vec{p}} > 0 \wedge hp_{\vec{p}} = \varnothing \\ bs_{\vec{p}}, & \text{otherwise} \end{cases}$$

$$ac'_{\vec{y}} = \begin{cases} ac_{\vec{y}} - 1, & \exists\, q : \vec{p} \bullet (bs'_q < bs_q \wedge `\vec{y}\text{'} \in \vec{a}_{ai_q}) \\ ac_{\vec{y}}, & \text{otherwise} \end{cases}$$

$$ai'_{\vec{p}} = \begin{cases} 0, & bs'_{\vec{p}} \leqslant 0 \\ ai_{\vec{p}}, & \text{otherwise} \end{cases}$$

$$up'_{\vec{y}} = \begin{cases} now', & up_{\vec{y}} = \infty \wedge \\ & \exists\, q : \vec{p} \bullet (bs_q > 0 \wedge bs'_q = 0 \wedge `\vec{y}\text{'} \in \vec{a}_{ai_q}) \\ up_{\vec{y}}, & \text{otherwise} \end{cases}$$

As before we assume that each conjunct containing a subscripted vector is duplicated with each vector element replacing all appearances of the vector. The tests '$\vec{y}$' $\in \vec{a}_{ai_q}$ denote a check to see if the *name* of the particular variable v in $\vec{y}$ is in the set of accessed variables.
As usual the tick action advances now. The other conjuncts control changes to the auxiliary variables in a pre-emptive execution environment.
The first says that the busy indicator bs_q for each process q in $\vec{p}$ is decremented, i.e., the process makes progress, only if there is no higher priority process currently active. For some process q, expression hp_q is the set of processes r from $\vec{p}$ currently performing an action A_{ai_r} with a priority higher than the action A_{ai_q} being performed by q.

$$hp_q = \{r : \vec{p} \mid ai_r \neq 0 \wedge (A_{ai_r} \mapsto A_{ai_q}) \in \mathcal{P}^+\}$$

The second conjunct says that the accessed indicator ac_v, for a variable v in $\vec{y}$, is decremented only if that variable is being accessed by the particular process q that is currently executing. The 'executing' process q is identified as the one that decremented its busy counter in this tick.
The third conjunct says that when any bs_q reaches zero, i.e., process q completes an action, then action identifier ai_q is reset to the null action.
The last conjunct sets up_v for any variable v in $\vec{y}$, that is being updated by some action, to the current absolute time whenever that action is completed. An action belonging to process q is known to have completed at the moment when bs_q becomes zero.

Multi-tasking makes it possible for more than one action to be *ready* on a given processor C at any one time. Thus the tick action takes the priority of actions into account when decrementing $bs_{\vec{p}}$ variables, and ensures that the highest priority ready process is the only one allowed to make progress. Also the $ai_{\vec{p}}$ identifiers are used so that the tick action knows whether to decrement the ac_v for a state variable v, because it is possible for time to pass without the process locking v making any progress. It is this technique that allows us

to avoid the maximal parallelism assumption so common in timed formalisms in the literature [KS93].

While an action A_i is executing, the value of each $up_{\vec{w}_i}$ variable is equal to the yet-to-be-defined value ∞, because potential pre-emption may mean that more than e time units of absolute time elapse between the time action A_i starts and finishes. This does not cause any anomalies, however, because no other action can access any $\vec{w}_i$ variable until A_i finishes, at which time the particular value of each $up_{\vec{w}_i}$ is fixed by the tick action. Thus no other action can 'see' the ∞ value.

Although quite complicated, such a detailed model allows us to capture intricacies of task scheduling not normally expressible in real-time formalisms. Context switching overheads, for instance, can be included by incorporating their duration in the $\mathbf{e}A_i$ expressions. Similarly, the overheads associated with a timer-driven scheduler, which uses regular clock interrupts to manipulate the task delay queue, can be added to the tick action.

For a scheduled action system to be a valid refinement of a process-based one the acceptable release times of low-priority actions may need to be further restricted to account for the fact that they may be pre-empted by higher-priority actions. (High-priority actions may also be blocked by lower-priority ones having locks on shared variables, but the atomicity of actions means that this form of interference should have already been considered in the design.) Importantly, the model developed above adheres to well-understood scheduling theory principles. The ability of each action to finish before its deadline can therefore be predicted using those schedulabilty tests applicable to static-priority pre-emptive scheduling [ABR+93].

4.5.1. Execution environment example

The physical mineshaft environment, and the hardware sensors and actuator, exist in true parallelism with our software system, so can be thought of as residing on their own *imaginary* processor.

$$RealWorld = \{Environment, Hardware\}$$

For the software we assume only one processor is available, which must be time-shared by the remaining processes.

$$Processor = \{MonitorWater, MonitorMethane, ControlPump\}$$

Priorities must be assigned to the actions that will execute on *Processor*. We make the reasonable assumptions that it takes longer to start or stop the pump than the allowable error in methane readings, and the lag in checking water levels is very small.

$$react > CH_4err$$
$$CH_4err > H_2Olag$$

Adopting *deadline monotonic* scheduling [ABR+93], where actions with shorter deadlines receive higher priority, then yields the following priority ordering.

$$\begin{aligned}\mathcal{P} = \{&CheckWater \mapsto CheckMethane,\\ &CheckMethane \mapsto PumpOff,\\ &CheckMethane \mapsto PumpOn\}\end{aligned}$$

The programmer is now obliged to prove that the system still has correct timing behaviour in the proposed scheduling regime. Since low-priority actions may be pre-empted, it may be necessary to reduce their available execution time by the duration of such pre-emptions. Actions *PumpOff* and *PumpOn* may be pre-empted by *CheckMethane*, but only if the methane pressure must be checked while one of them is responding to a change in water level. This can happen at most once, however, because of the timing property $react < CH_4per - CH_4err$, so the value of *POmax* need be reduced by only *CMmax*. This tells us that for feasibility of the design we require the time constants to obey the following property.

| $POmax - POmin \geqslant CMmax$

Similarly, action *CheckMethane* itself can be pre-empted by *CheckWater*. Since the direction in which the water level is moving can change only due to the pump being switched on or off, as shown by action *WaterLevel*, the number of these pre-emptions is again limited to one, because *PumpOff* and *PumpOn* cannot occur while *CheckMethane* is executing. Thus constant *CMmax* must be reduced by H_2Olag, which again dictates that our values of these constants must obey the following property for the design to be feasible.

| $CMmax - CMmin \geqslant H_2Olag$

At this level of detail the guard on the implicit tick action is changed so that processors do not idle unnecessarily. Whereas all action guards have required upper timing bounds so far, these may now be removed because actions now occur as early as possible. Indeed almost all references to time variables in guards should now be verifiably removable because program code cannot be generated for them. In formal development of real-time systems it is typical for abstract specifications to be expressed in terms of *absolute* time, and detailed implementable designs to be expressed in terms of times *relative* to significant events. The formal development procedure, and use of schedulability tests, must prove that the latter design will satisfy the former specification.

The *PumpOff* action must be shown to complete within its deadline. We have already done this above by observing that **e***PumpOff* units of processor time will be made available to the action before its deadline, despite possible pre-emption by action *CheckMethane*. The guard for *PumpOff* can thus be simplified to remove direct references to time.

g*PumpOff*
Danger; *Depth*; *PumpStatus*
---
pumpon
$\neg\ deep \lor shutdown$

This guard now refers only to state variables and can thus be easily implemented in a programming language. A similar simplification applies to **g***PumpOn*.

However, **g***CheckMethane* is left unchanged. It retains its references to time because, as shown in Section 4.6.1, time-dependent target language statements *are* used to achieve its periodic timing goal.

4.6. Mapping to target language

By following established real-time software engineering principles, the above methodology can result in designs directly expressible in a modern real-time programming language. In particular, the Real-Time Annex of the Ada 95 standard includes features that support scheduling theory concepts [II94]. It has a default static-priority pre-emptive scheduling policy, supports mutually-exclusive access to shared variables through its protected object construct, allows the programmer to directly express the allocation of priorities to tasks and shared variables, and supports priority inheritance.

Typically each application process in our final action system design will be either *periodic* or *sporadic*. A periodic process P_i is invoked regularly, every T_i time units. Each invocation must complete a number of distinct actions A_1 to A_m before some relative deadline D_i passes [ABR+93]. Each such action must therefore have a guard expressible in the following form.

$\mathbf{g}A_i$
$\ldots$
$\exists\, n : \mathbb{N} \bullet now \in n * T_i \mathrel{..} (n+1) * T_i$
B_i

An application-specific boolean condition B_i further restricts the circumstances under which the action may occur. (In most applications this would be used to ensure that each action is enabled only once in each period.) Notice that the deadline D_i is not directly mentioned. It must already have been proven, via the schedulability analysis of the overall design, that this time-consuming action will *always* begin and end within D_i time units of $n * T_i$, despite interference from other actions of equal or higher priority.

Invocations of sporadic processes S_j are released by the occurrence of some significant event, rather than the passage of time. Such events are defined by changes to the value of a shared variable (or group of variables) s. Typically there will be a *barrier* action A_1 awaiting some boolean condition B_s on the shared variable.

$\mathbf{g}A_1$
$\ldots$
B_s

(Statement $\mathbf{s}A_1$ would normally change the value of s in such a way that action A_1 disables itself.) Once such an action occurs, an invocation of the sporadic process will perform a number of further actions A_2 to A_m on local variables before some deadline D_j passes [ABR+93]. The timing constraints on these actions need not be explicitly stated because, again, it should already have been proven via schedulability analysis that actions A_1 to A_m always complete within D_j time units of B_s becoming true, in spite of any blocking or pre-emption by other actions.

We now outline the structure of the generic Ada 95 code. Naturally processes map to Ada 95 *tasks*. However where actions in different processes share variables, mutual exclusion must be enforced. Actions that access shared variables must be separated into groups R_s, one for each such shared 'resource' s, i.e., con-

venient grouping of shared variables. These groups then map to Ada 95 *protected objects*. Some action A_i, belonging to a process that accesses a shared variable, is replaced in the corresponding task with a call to a subroutine defined in the protected object.

Some global declarations import the standard Ada 95 real-time package and direct the compiler to use the static-priority scheduling policy.

```
with Ada.Real_Time; use Ada.Real_Time;
pragma Task_Dispatching_Policy(FIFO_Within_Priorities);
```

A periodic process P_i, consisting of actions A_1 to A_m which are all released in each period, is then implemented as a task with a delay statement to control the frequency of execution [BW95].

```
task P_i is
  pragma Priority(...); -- chosen to respect 𝒫 ordering
end P_i;

task body P_i is
  -- Local variables declared here
  Next: Time := Clock; -- specification time '0'
begin
  loop
    delay until(Next); -- wait for start of period
    while gA_1 or ··· or gA_m loop
      if gA_1 then sA_1
        ⋮
      elseif gA_m then sA_m
      end if;
    end loop;
    Next := Next + T_i; -- starting time of next period
  end loop;
end P_i;
```

The outer **loop** and **delay until** statement allow the task to proceed every T_i time units. The inner **while** loop and nested **if** statements implement a (deterministic) choice of those actions A_1 to A_m enabled at each task invocation. (This code is unnecessary if the same sequence of actions is always performed in each period.) The executable code corresponding to each $\mathbf{s}A_i$ is either included in-line or, if the action accesses variables shared between tasks, appears as a call to a protected subroutine.

Information about the required execution times of each action must be retained to guide later refinement of the individual actions. The timing requirements have not been fully discharged until satisfactory execution times for each action have been proven. Refinement of each $\mathbf{s}A_i$ to executable sequential code can proceed using timed extensions to conventional, sequential refinement rules [Fid94a, Fid94b, HU96].

At this level of detail we can account for the run-time overheads associated with the "outer" loop and selecting an action. So far it has been assumed that the overhead of evaluating $\mathbf{g}A_i$, and selecting the corresponding $\mathbf{s}A_i$ for execution, was incorporated in execution time $\mathbf{e}A_i$. This overhead can now be calculated

from the code fragments shown above [CBW94, Fid94b], and the available remaining execution time for the individual actions correspondingly reduced.

Each resource R_s requires a protected object to encapsulate the shared variable(s) s.

```
protected R_s is
  pragma Locking_Policy(Ceiling_Locking);
  pragma Priority(...); -- chosen to respect P ordering
  procedure A_update(...); -- action that writes to s
  function A_read return ...; -- action that reads value of s
  entry A_await(...); -- action that waits for a condition on s
    ⋮
private
  s : ...; -- declare shared variable(s)
end R_s;

protected body R_s is

  procedure A_update(...) is
  begin
    sA_update
  end A_update;

  function A_read return ... is
  begin
    return ...;
  end A_read;

  entry A_await(...) when gA_await is
  begin
    sA_await
  end A_await;
    ⋮
end R_s;
```

Subroutines declared within the protected object implement actions that access the shared variables. Some action A_{update} that modifies the shared variables is implemented as a procedure, called by the task that implements the process owning the action. Similarly, an action A_{read} that examines, but does not modify, the shared variables is implemented as a protected function. Sporadic tasks that await some condition on a shared variable to become true are implemented using entry calls. An action A_{await} from a sporadic task is implemented as an entry which can occur only when $\mathrm{g}A_{await}$ is true. As usual, it is good programming practice to make the actions performed by any of these subroutines as brief as possible to avoid unnecessarily blocking other tasks requiring access to shared variables. Ada 95 protected objects may also include interrupt handlers as a way of directly interfacing with hardware [II94].

Each sporadic process S_j, consisting of a barrier action A_1 that accesses a shared variable, and a number of subsequent actions A_2 to A_m on local vari-

ables, is then implemented by waiting for the condition on the shared variable to become true.

```
task Sj is
  pragma Priority(...); -- chosen to respect P ordering
end Sj;

task body Sj is
  -- Local variables declared here
begin
  loop
    A1(...); -- entry call
    while gA2 or ··· or gAm loop
      if gA2 then sA2
         ⋮
      elseif gAm then sAm
      end if;
    end loop;
  end loop;
end Sj;
```

4.6.1. Mapping example

The specification of *Processor* from Section 4.5.1 can now be mapped to Ada 95 code. In doing so timing analysis techniques can be applied to the code segments generated for each action A_i, in order to discharge the timing obligations relating to execution times $\mathbf{e}A_i$. Elsewhere we have discussed suitably analysable sequential Ada code subsets [CBW94].

Protected objects need to be constructed for the shared variables. As shown by the partitioning in Section 4.4.1, the *ControlPump* process shares variables *deep* and *shutdown* with both *MonitorWater* and *MonitorMethane*. We therefore group these two variables into an object `Warnings` with suitable subroutines for accessing them as required.

```
protected Warnings is
  pragma Locking_Policy(Ceiling_Locking);
  pragma Priority(Interrupt_Priority'First); -- high
  procedure CH4High;
  procedure CH4Low;
  entry Start;
  entry Stop;
private
  procedure CheckWater;
  pragma Attach_Handler(CheckWater,...);
  deep : Boolean := False;
  shutdown : Boolean := False;
end Warnings;

protected body Warnings is
```

```
procedure CH4High is -- part of sCheckMethane
begin
  shutdown := True;
end CH4High;

procedure CH4Low is -- part of sCheckMethane
begin
  shutdown := False;
end CH4Low;

procedure CheckWater is -- implements sCheckWater
begin
  deep := not deep;
end CheckWater;

entry Stop when not deep or shutdown is -- from gPumpOff
begin
  null;
end Stop;

entry Start when deep and not shutdown is -- from gPumpOn
begin
  null;
end Start;

end Warnings;
```

The boolean `shutdown` variable is modified by procedures `CH4High` and `CH4Low` which set it to `True` and `False` respectively.

We give no separate code for the simple *MonitorWater* process, assuming it is implemented by a hardware device attached to the mineshaft wall which generates an interrupt whenever the rising or falling water level passes a certain point. All that is then required is for interrupt handler `CheckWater` to toggle `deep` whenever such an interrupt occurs. This interrupt occurs at the highest priority, so it is easy to determine whether `CheckWater` is completed within H_2Olag time units, as required.

Entries `Stop` and `Start` act as the interface between the shared variables and the *PumpOff* and *PumpOn* actions. They allow these actions to test the values of the shared variables, and implement *part* of the guards for these two actions.

The *MonitorMethane* process is implemented as the following Ada 95 task.

```
task MonitorMethane is
  pragma Priority(Priority'First+1); -- medium
end MonitorMethane;

task body MonitorMethane is
  CH4gauge : Integer; -- must be initially less than CH4max
  for CH4gauge use ...;
  CH4Per : Time_Span := CH4per;
  Next : Time := Clock;
begin
  loop
```

```
      delay until Next; -- as per gCheckMethane
      if CH4gauge > CH4max then -- as per sCheckMethane
        Warnings.CH4High;
      else
        Warnings.CH4Low;
      end if;
      Next := Next + CH4per;
    end loop;
  end MonitorMethane;
```

Variable `CH4gauge` is assumed to be a memory-mapped i/o location directly linked to an analogue-to-digital converter attached to the methane sensing hardware.

Notice that there is no mention of durations CH_4err, *CMmin* or *CMmax* in this code. We are therefore obliged to prove that this code executes within these limits before this development step can be considered valid. This could be done experimentally but, in a safety-critical context, more formal approaches [CBW94, Fid94b] are preferable. We must prove that the code within the loop takes between *CMmin* and *CMmax* time units of *relative* processor time to execute. (Proof that the actions occur correctly in *absolute* time has already been outlined in Section 4.5.1. After schedulability analysis, the only timing obligations left should be on execution times.)

Finally the *ControlPump* process is implemented as the following task.

```
task ControlPump is
  pragma Priority(Priority'First); -- low
end ControlPump;

task body ControlPump is
  pumpon : Boolean := False;
begin
  loop
    if pumpon then -- half of gPumpOff and gPumpOn
      Warnings.Stop; -- half of gPumpOff
      pumpon := False; -- implements sPumpOff
      -- Turn the physical pump off
    else
      Warnings.Start; -- half of gPumpOn
      pumpon := True; -- implements sPumpOn
      -- Turn the physical pump on
    end if;
  end loop;
end ControlPump;
```

Here the two actions *PumpOff* and *PumpOn* have been implemented in one segment of sequential code. The **if** statement chooses between them. Since actions *PumpOff* and *PumpOn* use variables that are local to both task `ControlPump` and resource `Warnings`, the guards are evaluated in two parts. As they rely on complementary values of `pumpon`, it is possible to check this variable first and then await changes to either `deep` or `shutdown`. Timing analysis for this task, to prove that it executes in *POMin* to *POmax* time units, must include not only

the executable statements in task `ControlPump` but also the time required for entries `Stop` and `Start` in protected object `Warnings`.

5. Related work

A previous attempt to add time to action systems [KS93] was limited to applications where the functional behaviour of the system could not be time-dependent. Our model allows both time-dependent functional behaviour, and state-dependent timing behaviour, to be expressed. Nor did the previous approach address the issues of pre-emptive, overlapping actions.

The concept of a 'tick' action has been proposed many times in the literature. In particular our use of this method was inspired by Abadi and Lamport's work in the area [AL94].

Our use of Z together with action systems is similar to the work of Evans, who embeds the action-system-like *Unity* model for concurrency in Z [Eva94]. The most striking difference is that Evans assumes the implicit pre-condition of an operation is the same as its guard. This has a dramatic syntactic benefit. The outer **do** loop and guards are unnecessary and the specification looks exactly like a standard Z specification; the difference lies only in the operational interpretation applied to the specification. A disadvantage, however, is that much greater care is required in refining operations. Since guards are implicit, it is easy to accidently change the overall system behaviour when weakening a pre-condition.

Other work has sought to extend Z for real-time system development by adding Real-Time Logic features [Fid92, Lan95]. In particular Lano's object-oriented Z^{++} extensions [Lan95] have much in common with the approach in this paper, including trace-based top-level specifications and a well-defined development procedure. The method does not introduce scheduling concepts, however.

The complexity of a multi-tasking model has meant that, to date, we have concentrated on development 'in the small'. Therefore our formal model could be gainfully complemented by established large-scale real-time development procedures. For instance, an influence on the mineshaft case study presented above was the design methodology advocated by Lewerentz and Lindner for embedded real-time systems [LL94]. They recommend clearly distinguishing the system under development from its environment, and introducing extra environmental detail only when, and if, it is needed.

Furthermore, although we uses processes to define the concurrent components of a design, they are not intended as a general modularisation mechanism for managing large-scale developments. Therefore a broadly-based real-time development approach such as DARTS/DA [Gom88], which defines subsystems and their interfaces, and allows graphical representations of the system structure, would also form a useful framework for our formalism.

6. Conclusion

We have described and demonstrated a way of using action systems to formally develop concurrent systems with hard real-time constraints. We have shown how overlapping time-consuming behaviours, and the practicalities of pre-emptive scheduling, can be modelled in a formalism founded in interleaved, atomic actions.

The model is complex. This is partly due to the inherit complexity of the pre-emptive scheduling application, but can also be attributed to limitations of the action system specification notation. Elsewhere we have demonstrated that the same modelling and development exercise can be performed much more succinctly by using a trace-based notation throughout the refinement [FUKH96]. The numerous auxiliary action-system variables introduced above, used to keep track of the times at which certain events occur, are unnecessary in a trace-based model because the total history of activities is always readily available. Nevertheless, trace-based specifications are unfamiliar to most programmers, whereas the action system approach has the significant pragmatic advantage of building on existing, widely-understood intuitions about state machines.

Acknowledgements

We wish to thank Mark Utting, Peter Kearney and the anonymous referees for their comments. This project is funded by the Information Technology Division of the Australian Defence Science and Technology Organisation. The research reported in this paper was undertaken while Andy Wellings was on sabbatical leave at the University of Queensland.

References

[ABR+93] N. Audsley, A. Burns, M. Richardson, K. Tindell, and A. Wellings. Applying new scheduling theory to static priority pre-emptive scheduling. *Software Engineering Journal*, 8(5):284–292, September 1993.

[AL94] M. Abadi and L. Lamport. An old-fashioned recipe for real time. *ACM Transactions on Programming Language and Systems*, 16(5):1543–1571, September 1994.

[Bac92] R.J.R. Back. Refinement of parallel and reactive programs. Technical Report Caltech-CS-TR-92-23, California Institute of Technology, 1992.

[Bac93] R.J.R. Back. Atomicity refinement in a refinement calculus framework. Technical Report 141, Abo Akademi, Dept. of Computer Science, March 1993.

[BS91] R.-J. Back and K. Sere. Stepwise refinement of action systems. *Structured Programming*, 12(1):17–30, 1991.

[BS94] R.J.R. Back and K. Sere. From action systems to modular systems. In M. Naftalin, T. Denvir, and M. Bertran, editors, *FME'94: Industrial Benefit of Formal Methods*, volume 873 of *Lecture Notes in Computer Science*, pages 1–25. Springer-Verlag, 1994.

[BvW94] R.J.R. Back and J. von Wright. Trace refinement of action systems. In B. Jonsson and J. Parrow, editors, *CONCUR'94: Concurrency Theory*, volume 836 of *Lecture Notes in Computer Science*, pages 367–384. Springer-Verlag, 1994.

[BW90] A. Burns and A.J. Wellings. *Real-Time Systems and their Programming Languages*. Addison-Wesley, 1990.

[BW95] A. Burns and A.J. Wellings. Engineering a hard real-time system: From theory to practice. *Software–Practice & Experience*, 25(7):705–726, July 1995.

[CBW94] R. Chapman, A. Burns, and A.J. Wellings. Integrated program proof and worst-case timing analysis of SPARK Ada. In *ACM Workshop on language, compiler and tool support for real-time systems*. ACM Press, 1994.

[Eva94] A. Evans. Specifying and verifying concurrent systems using Z. In M. Naftalin, T. Denvir, and M. Bertran, editors, *FME'94: Industrial Benefit of Formal Methods*, volume 873 of *Lecture Notes in Computer Science*, pages 366–380. Springer-Verlag, 1994.

[Fid92] C.J. Fidge. Specification and verification of real-time behaviour using Z and RTL. In J. Vytopil, editor, *Formal Techniques in Real-Time and Fault-Tolerant Systems*, volume 571 of *Lecture Notes in Computer Science*, pages 393–409. Springer-Verlag, 1992.

[Fid94a] C.J. Fidge. Adding real time to formal program development. In M. Naftalin, T. Denvir, and M. Bertran, editors, *FME'94: Industrial Benefit of Formal Methods*, volume 873 of *Lecture Notes in Computer Science*, pages 618–638. Springer-Verlag, 1994.

[Fid94b] C.J. Fidge. Proof obligations for real-time refinement. In D. Till, editor, *Sixth Refinement Workshop*, pages 279–305. Springer-Verlag, 1994.

[FUKH96] C.J. Fidge, M. Utting, P. Kearney, and I.J. Hayes. Integrating real-time scheduling theory and program refinement. In M.-C. Gaudel and J. Woodcock, editors, *FME'96: Industrial Benefit and Advances in Formal Methods*, volume 1051 of *Lecture Notes in Computer Science*, pages 327–346. Springer-Verlag, 1996.

[Gom88] H. Gomaa. Extending the DARTS software design method to distributed real time applications. In *Proc. 21st Hawaii Int. Conf. on System Sciences*, pages 252–261, 1988.

[HU96] I.J. Hayes and M. Utting. Coercing real-time refinement: A transmitter. In *Proc. Northern Formal Methods Workshop*, Ilkley, U.K., September 1996. To appear.

[II94] International Organization for Standardization and International Electrotechnical Commission. *Ada Reference Manual: Language and Standard Libraries*, 6.0 edition, December 1994. International Standard ISO/IEC 8652:1995.

[KS93] R. Kurki-Suonio. Stepwise design of real-time systems. *IEEE Trans. Software Engineering*, 19(1):56–69, January 1993.

[Lan95] K. Lano. Reactive system specification and refinement. In P. Mosses et al., editors, *TAPSOFT'95: Theory and Practice of Software Development*, volume 915 of *Lecture Notes in Computer Science*, pages 696–710. Springer-Verlag, 1995.

[LB90] A. Lister and A. Burns. An architectural framework for timely and reliable distributed information systems (TARDIS): Description and case study. Technical Report YCS 140, Department of Computer Science, University of York, August 1990.

[LL94] C. Lewerentz and T. Lindner. Formal methods for reactive systems. In *Tutorial Proceedings: FME'94 Symposium*, October 1994.

[MH92] B.P. Mahony and I.J. Hayes. A case-study in timed refinement: A mine pump. *IEEE Transactions on Software Engineering*, 18(9):817–826, September 1992.

[Pil91] M.J. Pilling. Dangers of priority as a structuring principle for real-time languages. *Australian Computer Science Communications*, 13(1):18–1–18–10, February 1991.

[PST91] B. Potter, J. Sinclair, and D. Till. *An Introduction to Formal Specification and Z*. Prentice-Hall, 1991.

[SRL90] L. Sha, R. Rajkumar, and J. Lehoczky. Priority inheritance protocols: An approach to real-time synchronization. *IEEE Trans. Computers*, 39(9):1175–1185, September 1990.

[SZH93] D. Scholefield, H. Zedan, and Jifeng He. Real-time refinement: Semantics and application. In A. Borzyszkowski and S. Sokolowski, editors, *Mathematical Foundations of Computer Science 1993*, volume 711 of *Lecture Notes in Computer Science*, pages 693–702. Springer-Verlag, 1993.