A model of real-time distributed systems

A. Wabenhorst

Software Verification Research Centre

University of Queensland QLD 4072, Australia

akw@it.uq.edu.au

Abstract

A model of real-time distributed systems using action systems is presented, with semantics given in terms of the predicate transformer semantics of non-timed action systems. An example is given to illustrate the model and to demonstrate its stepwise development from specification. A strength of the approach presented is the wide range of real-time systems which can be specified in the model.

Keywords

Real time, action systems, weakest preconditions, refinement, formal methods

1 INTRODUCTION

The problem of specifying real-time distributed systems has occupied computer scientists in recent times. This is partly because real-time systems, particularly in the area of defence, are more often safety-critical than systems without the concept of time. Real-time systems often need to react to an environment which may not always be controllable. In such cases, validation by testing may be particularly unreliable, and there is greater scope for the use of formal methods.

Real-time systems reacting to an environment present the challenge of modelling the environment and its interaction with the system, and showing that the system meets its requirements when reacting to the environment. A particular difficulty is modelling time for different processes acting in parallel, and the issue arises as to whether there is the concept of global time.

For our model, we choose the notation of action systems (Back 1989) for their developmental clarity and general expressive power, particularly of parallelism, and because their predicate transformer semantics are concise and calculational.

We begin in Section 2 by recalling the definitions of predicate transformers (Dijkstra *et al.* 1990) and action systems (Back 1989), together with the

resulting definition of refinement. We define action sequences and give a characterisation of the semantics of iteration using them. In Section 3, we motivate and explain our model of real-time action systems and state some of its properties. The model is illustrated by a message passing example in Section 4. Finally in Section 5, we discuss our results and relate them to those of others.

2 BACKGROUND

2.1 Predicate transformers

Let $\Sigma_{\mathcal{A}}$ denote the state space of a program $\mathcal{A}$ expressed in the language of guarded commands (Dijkstra *et al.* 1990). A *predicate* is a function $f : \Sigma_{\mathcal{A}} \rightarrow \mathbb{B}$, the set of all of which will be denoted by $Pred\mathcal{A}$.

A statement S is a predicate transformer $S : Pred\mathcal{A} \rightarrow Pred\mathcal{A}$ known as the *weakest precondition* (Dijkstra *et al.* 1990) satisfying, for all $P, Q : Pred\mathcal{A}$, the axioms

- monotonicity: if $P \Rightarrow Q$, then $S(P) \Rightarrow S(Q)$;
- conjunctivity: if $I \neq \emptyset$ is an index set, then

$$(\forall \alpha \in I \ S(P_\alpha)) \Rightarrow S(\forall \alpha \in I \ P_\alpha).$$

Informally, we can regard the weakest precondition $S(P)$ for $P : Pred\mathcal{A}$ to be the weakest predicate on the variable states from which execution of the statement S guarantees termination in a state satisfying P . In particular, $S(true)$ is the weakest predicate which guarantees that S terminates. Notice that we allow statements which do not satisfy strictness $S(false) \Leftrightarrow false$.

Statements and their associated weakest precondition transformers include those in Figure 1 (Boon 1982, Dijkstra *et al.* 1990) and the specification statements of Morgan (1994).

An *action system* (Back 1989) has the form

```

var            $x_1, \dots, x_m$ 
init          $x_1, \dots, x_m := X_1, \dots, X_m$ 
do    $\prod_{i=1}^n A_i$  od,
```

where $x_1, \dots, x_m$ are typed variables and $A_1, \dots, A_n$ are actions. Sometimes we will omit the variable declaration when it can be deduced from the body of the action system. When a variable is initialised arbitrarily within its type, we will omit an explicit initialisation.

Definition 1 (Refinement) A statement S is refined by a statement S' (notation $S \leq S'$) if for all $P : Pred\mathcal{A}$, $S(P) \Rightarrow S'(P)$. $\square$

-
- skip: $\mathbf{skip}(P) \triangleq P$.
 - assertion: $(!b)(P) \triangleq b \wedge P$.
 - abort: $\mathbf{abort} \triangleq !false$.
 - assignment: $(\vec{x} : \in \vec{X})(P) \triangleq \forall \vec{y} \in \vec{X} P[\vec{y}/\vec{x}]$, where $\vec{x}$ is a list of variables and $P[\vec{y}/\vec{x}]$ is the predicate P with each member of $\vec{y}$ replacing the corresponding member of $\vec{x}$.
 - sequential composition: $(S_1; S_2)(P) \triangleq S_1(S_2(P))$.
 - choice: $(S_1 \parallel S_2)(P) \triangleq S_1(P) \wedge S_2(P)$.
 - actions: $(g \rightarrow S)(P) \triangleq \neg g \vee S(P)$. Here, g is called the *guard* of the action.
 - alternation: $(\mathbf{if} \prod_{i=1}^n g_i \rightarrow S_i \mathbf{fi})(P) \triangleq \bigvee_{i=1}^n g_i \wedge (\prod_{i=1}^n g_i \rightarrow S_i)(P)$.
 - iteration: $(\mathbf{do} \prod_{i=1}^n g_i \rightarrow S_i \mathbf{od})(P) \triangleq \exists \kappa : \text{Ord } H_\kappa(P)$, where Ord is the class of ordinal numbers and

$$\begin{aligned}
 H_0(P) &\triangleq P \wedge \neg \bigvee_{i=1}^n g_i, \\
 H_{\kappa+1}(P) &\triangleq H_0(P) \vee (\mathbf{if} \prod_{i=1}^n g_i \rightarrow S_i \mathbf{fi})(H_\kappa(P)) \text{ and} \\
 H_\delta(P) &\triangleq \exists \kappa < \delta H_\kappa(P), \text{ where } \delta \text{ is a limit ordinal.}
 \end{aligned}$$

Figure 1 Some statements and their weakest preconditions.

2.2 Action sequences

For an action system $\mathcal{A}$, $\text{seq}\mathcal{A}$ denotes the set of all finite sequential compositions of actions in $\mathcal{A}$, while the elements of the set $\text{seq}_\omega\mathcal{A}$ are written as sequential compositions of infinitely many actions, but do not have predicate transformer semantics. These two sets are the *action sequences* of $\mathcal{A}$. For an action sequence s of length at least m , $s|_m$ is the prefix of s with length m . The predicate transformer **skip** is considered to be an action sequence of length 0. For an action A , A^m is the sequential composition of A with itself m times.

The following characterisation (Morris 1987, Butler *et al.* 1995, Wabenhorst 1996) of the weakest precondition of an action system can be thought of as a decomposition. The first conjunct is a safety condition on the finite action sequences, also known as the weakest errorfree precondition (Hesselink 1995), which states that if $\mathcal{A}$ terminates at all, then it terminates in state P . The second conjunct is a liveness condition on the infinite action sequences which states that $\mathcal{A}$ terminates eventually.

Theorem 1 *Let $\mathcal{A} \triangleq \mathbf{do} \prod_{i=1}^n g_i \rightarrow S_i \mathbf{od}$ be an action system. Then for all $P : \text{Pred}\mathcal{A}$,*

$$\mathcal{A}(P) \Leftrightarrow \forall s : \text{seq}\mathcal{A} \ s(P \vee \bigvee_{i=1}^n g_i) \wedge \mathcal{A}(\neg \bigvee_{i=1}^n g_i)$$

and $A(\neg \bigvee_{i=1}^n g_i) \Leftrightarrow \exists \kappa \mathcal{H}_\kappa^*$, where

$$\begin{aligned} \mathcal{H}_0^* &\triangleq \neg \bigvee_{i=1}^n g_i \\ \mathcal{H}_{\kappa+1}^* &\triangleq \forall s : seq_\omega \mathcal{A} \exists m < \omega \ s|_m(\mathcal{H}_\kappa^*) \\ \mathcal{H}_\delta^* &\triangleq \exists \kappa < \delta \ \mathcal{H}_\kappa^*. \end{aligned}$$

Proof:

See Wabenhorst (1996). □

2.3 The Movers Theorem

Sufficient conditions under which an iteration within an action system can be flattened are given in the Movers Theorem (Back 1989, Wabenhorst 1997), which will be used extensively in the example of Section 4.

Theorem 2 (Movers Theorem) *Let*

$$\begin{array}{ll} \mathcal{A} \triangleq & \text{init;} \\ & \text{do} \quad \begin{array}{l} g \rightarrow S \\ \parallel_{j=1}^m h_j \rightarrow T_j \end{array} \\ & \text{od} \end{array}, \quad \begin{array}{ll} \mathcal{B} \triangleq & \text{init;} \\ & \text{do} \quad \begin{array}{l} g \rightarrow S_0 \\ \parallel_{i=1}^n g_i \rightarrow S_i \\ \parallel_{j=1}^m h_j \rightarrow T_j \end{array} \\ & \text{od} \end{array}$$

and $\mathcal{C} \triangleq \text{do } \parallel_{i=1}^n g_i \rightarrow S_i \text{ od}$ such that $!(\neg \bigvee_{i=1}^n g_i); g \rightarrow S \leq g \rightarrow S_0; \mathcal{C}$.

Then $\mathcal{A} \leq \mathcal{B}$, if the following conditions hold:

- 1.(a) For all j such that $1 \leq j \leq m$, $\bigvee_{i=1}^n g_i \Rightarrow (h_j \rightarrow T_j)(\bigvee_{i=1}^n g_i)$.
- (b) For all j such that $1 \leq j \leq m$, $\neg \bigvee_{i=1}^n g_i \Rightarrow (h_j \rightarrow T_j)(\neg \bigvee_{i=1}^n g_i)$.
- (c) $\text{init}(\neg \bigvee_{i=1}^n g_i) \Leftrightarrow \text{true}$.
2. $\bigvee_{i=1}^n g_i \Rightarrow \neg g$.
3. The set $\{h_j \rightarrow T_j \mid 1 \leq j \leq m\}$ can be expressed as the disjoint union of sets $\mathcal{L}$ (called the left movers) and $\mathcal{R}$ (called the right movers), such that:
 - (a) For all $L \in \mathcal{L}$, $L; g \rightarrow S_0 \leq g \rightarrow S_0; L$.
 - (b) For all $L \in \mathcal{L}$ and i such that $1 \leq i \leq n$, $L; g_i \rightarrow S_i \leq g_i \rightarrow S_i; L$.
 - (c) For all $L \in \mathcal{L}$ and $R \in \mathcal{R}$, $L; R \leq R; L$.
 - (d) For all $R \in \mathcal{R}$ and i such that $1 \leq i \leq n$, $g_i \rightarrow S_i; R \leq R; g_i \rightarrow S_i$.
 - (e) $\bigvee_{i=1}^n g_i \Rightarrow (\text{do } \parallel_{R \in \mathcal{R}} R \text{ od})(\text{true})$.

- (f) All left movers and all $g_i \rightarrow S_i$, $1 \leq i \leq n$, are boundedly nondeterministic.

□

Other refinement theorems include Monotonicity (Back 1989, Wabenhorst 1996), which permits the refinement of actions one-by-one with the result that the action system as a whole is refined also.

3 A MODEL OF TIME

3.1 Assumptions

Consider the action system

$$\begin{array}{lll} \mathcal{A} \triangleq & \text{var} & \dots \\ & \text{init} & \dots \\ & \text{do} & \prod_{i=1}^M \prod_{j=1}^{N_i} g_{i,j} \rightarrow S_{i,j} \\ & \text{od} & \end{array}$$

which we wish to transform into a timed action system. The subscript i in the action $g_{i,j} \rightarrow S_{i,j}$ denotes the process which executes the action, while j labels the action within process i . As in other models of time, actions in different processes may be executed concurrently if they do not interfere, otherwise one or more actions must wait. Actions within the same process are executed in sequence.

We make the assumption of maximal parallelism: if a process is idle and able to execute one of its actions, then it does so immediately. In any case, we can add actions which explicitly delay the execution of other actions if we desire.

We propose the syntax for the timed action system corresponding to $\mathcal{A}$ in Figure 2. Here, we use dashes — to delineate processes and introduce a time variable t_i of type *Time*, which can be $\mathbb{R}$ or $\mathbb{Z}$, for each process i . The variable t_i contains the time at which the action last executed by process i has ceased executing; therefore this is the time at which the next action may be executed by process i . The assignment $t_i := \dots$ updates t_i to take into account the time that the corresponding action takes, possibly depending on the states of other variables. The assignment might involve a nondeterministic choice, and could take into account the time taken to make a scheduling decision. The sets $\text{mutex}_{i,j}$ contain the names of actions which interfere with action (i, j) ; these are the responsibility of the developer. We use the notation **do-rt** ... **od-rt** to signal that the semantics do not correspond to those of a

```

var          ...
init         ...
do-rt   $\underbrace{\prod_{j=1}^{N_1} g_{1,j} \textbf{ without } mutex_{1,j} \rightarrow S_{1,j}; t_1 := \dots}_{\dots} (1, j)$ 
           $\underbrace{\prod_{j=1}^{N_2} g_{2,j} \textbf{ without } mutex_{2,j} \rightarrow S_{2,j}; t_2 := \dots}_{\dots} (2, j)$ 
          ...
           $\underbrace{\prod_{j=1}^{N_M} g_{M,j} \textbf{ without } mutex_{M,j} \rightarrow S_{M,j}; t_M := \dots}_{\dots} (M, j)$ 
od-rt

```

Figure 2 Syntax for the general real-time action system.

standard action system. Otherwise, the syntax is the same as for the standard action system $\mathcal{A}$.

Our task now is to define the semantics of the real-time action system in Figure 2. To this end, we will obtain a standard action system incorporating as design decisions the assumptions made in this section. We will then define the semantics of the real-time action system to be the semantics of this standard action system.

3.2 Time variables

To the action system $\mathcal{A}$ in Section 3.1, we add the time variables t_i of Figure 2. Thus, the time variables are initialised to 0:

init $\dots; \prod_{i=1}^M t_i := 0,$

where the ellipses correspond to the initialisation of the original action system $\mathcal{A}$. Also, the time variables are updated during execution to take into account the time taken by an executing action: thus, $g_{i,j} \rightarrow S_{i,j}$ is modified to become $g_{i,j} \rightarrow S_{i,j}; t_i := \dots$.

We consider the current global time of an action system to be

$$T \triangleq \min_{k=1}^M t_k.$$

This can be regarded as an invariant which connects the real world with the action system. Here, T is an abbreviation designed to simplify the exposition, rather than a variable which would result in extra assignment statements in the action system.

At the current global time T , there is some process which is ready to execute an action. This is because there must be at least one i for which $t_i = T$. Indeed, there may be more than one, such as upon initialisation when all time variables t_i are 0. In accordance with the semantics of action systems, a nondeterministic choice is made between the enabled actions in such processes, and the chosen action executes subsequently. The time variable of the process will be updated, which may or may not increase the current global time.

Usually the time variables will be different from each other, reflecting the fact that some processes are in advance of the current global time. However, we stipulate that the next action chosen for execution may only occur in a process which is at the current global time; this limits the extent to which one process may be in advance of other processes. Thus, we strengthen the guard of each action (i, j) as follows:

$$g_{i,j} \wedge t_i = T \rightarrow S_{i,j}; t_i := \dots$$

3.3 Blocking actions

The concurrent execution of actions in different processes is achieved by the usual interleaving semantics of action systems. However, when interference would occur, concurrent execution must be prevented.

Recall from Section 3.2 that the new choice of an action (say b in process i) to execute in the action system may only be made at the current global time T . Suppose that previously, action a (in process m different from i) was chosen for execution, so that the update of t_m has resulted in m being ahead of the current global time. The choice of b would correspond, in the real world, to a and b executing concurrently. However in the case of interference between a and b , this should be prevented in accordance with our assumptions in Section 3.1. As a was chosen for execution previously, it must be b which is prevented from executing.

We introduce new variables which help us to deduce when interference between actions would occur. Thus, we define $mutex_b$ for each action b to be the set of names of all other actions which prevent execution of b because of interference. Normally, two actions are considered to interfere if one accesses a variable which the other writes. Also, we define $exec_i$ for each process i to be the last action chosen for execution in process i .

We assume that the potential for interference between actions does not change at run time. It could be argued that an action a with body

```
if  $g \rightarrow x := 1 \parallel \neg g \rightarrow y := 2$  fi
```

interferes with an action b which accesses y , only if $\neg g$ holds. However, it may not be possible for b to evaluate g at run time because g might depend

on variables which are either local to a 's process or are being changed by a (such as x and y), and hence are not accessible to b . Thus, $mutex_b$ is set upon initialisation of the action system by analysing the syntax of the original action system $\mathcal{A}$, and not changed during execution.

init $\dots; \prod_{i=1}^M t_i := 0; \prod_{i=1}^M \prod_{j=1}^{N_i} mutex_{i,j} := \dots$

Using the new variables just introduced, we deduce that an action $b = (i, j)$ would be prevented from execution by interference with action $a = (m, n)$ at the current global time T if

$$t_i < t_m \wedge exec_m \in mutex_{i,j},$$

and we strengthen the guard of action (i, j) from Section 3.2 to take this into account. As a result the guard, which we abbreviate to $h_{i,j}$, becomes

$$h_{i,j} \triangleq g_{i,j} \wedge t_i = T \wedge \bigwedge_{m=1}^M (t_i < t_m \Rightarrow exec_m \notin mutex_{i,j}).$$

We also update $exec_i$ in the body of (i, j) , so (i, j) becomes

$$h_{i,j} \rightarrow S_{i,j}; t_i, exec_i := \dots, (i, j).$$

The $exec_m$ variables may have arbitrary values initially, so they need not be initialised explicitly. This is because the value of $exec_m$ is not needed in the guard of (i, j) until $t_i < t_m$. In this case, the body of the action in process m currently executing will have updated $exec_m$.

3.4 The wait action

When a choice of an action to be executed is required at the current global time, each possible action might be blocked. This could occur either because its original guard $g_{i,j}$ is false, or because of potential interference with actions executing in other processes (see Section 3.3). If all processes are at the current global time, then there is no potential interference and termination has occurred. However, if some processes are ahead of the current global time, then increasing the current global time (that is, *waiting*) might result in at least one action becoming unblocked. The earliest when this might occur is the earliest completion time of an action after the current global time, that is

$$NT \triangleq \min_{\{m | t_m > T\}} t_m.$$

```

var          ...;  $\prod_{i=1}^M t_i : \text{Time}; \prod_{i=1}^M \text{exec}_i : \text{ActionName};$ 
                $\prod_{i=1}^M \prod_{j=1}^{N_i} \text{mutex}_{i,j} : \mathbb{P}(\text{ActionName})$ 

init         ...;  $\prod_{i=1}^M t_i := 0; \prod_{i=1}^M \prod_{j=1}^{N_i} \text{mutex}_{i,j} := \dots$ 

do    $\prod_{i=1}^M \prod_{j=1}^{N_i} h_{i,j} \rightarrow S_{i,j}; t_i, \text{exec}_i := \dots, (i, j) \quad (i, j)$ 
       $\square$        $(\bigvee_{i=1}^M t_i \neq T) \wedge \bigwedge_{i=1}^M \bigwedge_{j=1}^{N_i} \neg h_{i,j}$ 
                $\rightarrow \prod_{i=1}^M (t_i = T \rightarrow t_i := NT) \quad (W)$ 

od

```

where

$$\begin{aligned}
 T &\triangleq \min_{k=1}^M t_k \\
 NT &\triangleq \min_{\{m \mid t_m > T\}} t_m \\
 h_{i,j} &\triangleq g_{i,j} \wedge t_i = T \wedge \bigwedge_{m=1}^M (t_i < t_m \Rightarrow \text{exec}_m \notin \text{mutex}_{i,j})
 \end{aligned}$$

Figure 3 Semantics for the general real-time action system.

With these considerations, we construct the wait action as follows:

$$\left(\bigvee_{i=1}^M t_i \neq T \right) \wedge \bigwedge_{i=1}^M \bigwedge_{j=1}^{N_i} \neg h_{i,j} \rightarrow \prod_{i=1}^M (t_i = T \rightarrow t_i := NT)$$

The condition for blocking is the conjunction of the negations of the guards of all actions, that is $\bigwedge_{i=1}^M \bigwedge_{j=1}^{N_i} \neg h_{i,j}$, where $h_{i,j}$ is as defined in Section 3.3. We exclude waiting when termination has occurred, that is, when there is no process executing past the current global time. For this reason, $(\bigvee_{i=1}^M t_i \neq T)$ is added to the guard. All time variables t_i at the current global time (that is, $t_i = T$) are updated to NT in the body of the wait action.

Note that the body of the wait action does not change exec_i . This is because exec_i is never used for t_i at the current global time. (See also the justification in Section 3.3 for exec_i not being initialised explicitly.)

3.5 Real-time action systems

The action system constructed in the previous sections is shown in Figure 3, and incorporates the desired properties of the real-time action system of Fig-

ure 2 described informally in Section 3.1. Therefore, we define the semantics of the real-time action system in Figure 2 to be the semantics of the action system in Figure 3.

We now state some properties of real-time action systems. The following lemma, which can be proven using predicate calculus, shows that the termination condition of the real-time action system is closely related to that of the corresponding standard action system.

Lemma 1 *The disjunction of the guards of the action system in Figure 3 is*

$$\left(\bigvee_{i=1}^M t_i \neq T \right) \vee \bigvee_{i=1}^M \bigvee_{j=1}^{N_i} g_{i,j}$$

□

One would expect the real-time action system to implement the standard action system $\mathcal{A}$ in Section 3.1. This is indeed the case:

Theorem 3 *The real-time action system in Figure 3 refines the standard action system $\mathcal{A}$.*

Proof:

See the Appendix. □

The real-time action system with only one process behaves exactly like the corresponding standard action system: the guards $h_{1,j}$ are $g_{1,j}$, and the guard of the wait action is false.

3.6 The environment

Normally, we would expect an action a to interfere with an action b in a different process exactly when the second interferes with the first, that is

$$a \in \text{mutex}_b \Leftrightarrow b \in \text{mutex}_a.$$

However, there may be circumstances in which the relaxation of this symmetry property is convenient. Consider a situation where action b models part of the environment and action a is in a different process which reacts to the environment. It could be argued that execution of b should not be delayed through a accessing the environment, but that if b is executing then a should be delayed. In this case we would have $a \in \text{mutex}_b$ but not $b \in \text{mutex}_a$.

```

var       $in, msg : seqData; inchar, last : Data \cup \{\perp\}; r : \mathbb{B}$ 
init      $inchar, last, msg, r := \perp, \perp, \emptyset, false$ 
do-rt     $in \neq \emptyset \wedge inchar = \perp \wedge \neg r \rightarrow$ 
            $in, inchar, r := tl(in), hd(in), true \quad (1, 1)$ 
            $\square \quad inchar \neq \perp \wedge r \rightarrow r, t_1 := false, t_1 + chdef \quad (1, 2)$ 
            $\square \quad inchar \neq \perp \wedge \neg r \rightarrow inchar, t_1 := \perp, t_1 + chsep \quad (1, 3)$ 


---


            $\square \quad inchar \neq \perp \wedge inchar \neq last \textbf{ without } \{(1, 3)\} \rightarrow$ 
            $msg, last, t_2 := msg \frown inchar, inchar, t_2 + chread \quad (2, 1)$ 
od-rt

```

Figure 4 Final message passing implementation.

The model as it stands results in the continued execution of $a = (m, n)$ after $b = (i, j)$ has been chosen for execution. However, it may be more appropriate to abort in these circumstances, so we could modify (i, j) to become

$$h_{i,j} \rightarrow ! \left(\bigwedge_{m=1}^M (t_i < t_m \Rightarrow (i, j) \notin mutex_{exec_m}) \right); S_{i,j}; t_i, exec_i := \dots, (i, j)$$

where an assertion $!P$, defined in Figure 1, skips if P holds and aborts otherwise. Modifying (i, j) in this way would mean that Theorem 3 would no longer hold in general, but would depend on satisfactory time updates $t_k := \dots$

4 EXAMPLE: MESSAGE PASSING

We now illustrate the concepts in previous sections by an example, where we achieve a final implementation from a high level specification by stepwise refinement using the Movers Theorem and other results.

In this example, based on that in Hayes *et al.* (1997), a message is constructed from character-by-character input. A character appears in *inchar* for a duration $chdef > 0$, during which time the character must be added to the message *msg*. The time required for this is $chread > 0$, which we stipulate to be no larger than $chdef$. After duration $chdef$, *inchar* is undefined for duration $chsep > 0$ before the next character appears. The characters appearing in *inchar* are taken from a fixed finite channel *in*, so that termination will be guaranteed. The process constructing the message *msg* has no access to *in*, but must rely on *inchar*.

The final implementation is shown in Figure 4, where the first process implements the environment and the second constructs the message. The boolean *r* holds if *inchar* is readable, and acts as a synchronisation variable which helps to ensure that the actions in the first process are always executed in the

order (1, 1), then (1, 2), then (1, 3). The action (1, 1) updates *inchar* from *in* instantaneously, as t_1 does not change; this technique is also used by Fidge *et al.* (1997). Here, the functions *hd* and *tl* give the head and tail of a channel respectively and $\perp$ is a special character which is different from the characters in *Data*. The action (1, 2) waits for duration *chdef*, during which time *inchar* may be accessed by the other process. The action (1, 3) represents *inchar* being undefined for time *chsep*. In action (2, 1), *last* is used to detect when a new character has arrived in *inchar*: *last* stores the character last received through *inchar*, which we compare with the current value of *inchar*. Here, we assume that consecutive values in *in* are different, if necessary by appending a label to each datum. Using *last*, we avoid assumptions about timing behaviour to detect when a new *inchar* has arrived.

We set $mutex_{2,1} = \{(1, 3)\}$ to prevent the second process from beginning to access *inchar* while *inchar* is no longer defined. It makes no difference whether $mutex_{2,1}$ contains (1, 1) or not, because (1, 1) has duration 0. For the environment process, *mutex* is empty because the environment is not affected by the second process.

4.1 Specification and first steps

As a high level specification,

$in, msg := \emptyset, in$

constructs the final message *msg* directly from *in*. This can be refined to

```

msg :=  $\emptyset$ ;
do   in  $\neq \emptyset \rightarrow in, msg := tl(in), msg \hat{\ } hd(in)$ 
od

```

using one of the techniques of Morgan (1994), for example Exercise 5.6.

The definition of refinement permits the introduction of new variables, so we add the variable *inchar* by initialising it to $\perp$ and adding the conjunct $inchar = \perp$ to the guard of the single action. This is then refined to

```

inchar, msg :=  $\perp, \emptyset$ ;
do   in  $\neq \emptyset \wedge inchar = \perp \rightarrow in, inchar := tl(in), hd(in)$ 
    [] inchar  $\neq \perp \rightarrow inchar, msg := \perp, msg \hat{\ } hd(in)$ 
od

```

as a straightforward application of the Movers Theorem.

Now the variable *last* is introduced: it is initialised to $\perp$ and $last := inchar$ is added to the body of the second action. Since $inchar \neq \perp \Rightarrow inchar \neq last$ is an invariant, $inchar \neq last$ may be added as a conjunct to the guard of the second action. Theorem 3 for one process is applied to obtain Figure 5.

```

inchar, last, msg := ⊥, ⊥, ∅;
do-rt    in ≠ ∅ ∧ inchar = ⊥ →
          in, inchar, t1 := tl(in), hd(in), t1 + chdef
    [] inchar ≠ ⊥ ∧ inchar ≠ last →
          msg, last, inchar, t1 := msg ∩ inchar, inchar, ⊥, t1 + chsep
od-rt

inchar, last, msg, t1 := ⊥, ⊥, ∅, 0;
do    in ≠ ∅ ∧ inchar = ⊥ →
          in, inchar, t1, exec1 := tl(in), hd(in), t1 + chdef, (1, 2)
    [] inchar ≠ ⊥ ∧ inchar ≠ last →
          msg, last, inchar, t1 := msg ∩ inchar, inchar, ⊥, t1 + chsep;
          exec1 := (1, 3)
od

```

Figure 5 One process.

4.2 Two processes

The next refinement step, which splits the single process into two, is the most delicate. First, we add the time variable t_2 by appending $t_2 := t_1$ at the end of the body of the second action. We can add conjuncts $t_1 \leq t_2$ and $t_2 \leq t_1 \wedge (t_2 < t_1 \Rightarrow \text{exec}_1 \neq (1, 3))$ to the guards of the first and second actions respectively because

$$\text{inchar} = \perp \Rightarrow t_1 = t_2 \quad \text{and} \quad \text{inchar} \neq \perp \Rightarrow (t_1 = t_2 + \text{chdef} \wedge \text{exec}_1 \neq (1, 3))$$

are invariants of the resulting action system. Now we use the Movers Theorem with only right movers and

```

t2 ≤ t1 ∧ (t2 < t1 ⇒ exec1 ≠ (1, 3)) ∧ inchar ≠ ⊥ ∧ inchar ≠ last →
  msg, last, inchar, t1 := msg ∩ inchar, inchar, ⊥, t1 + chsep;
  t2, exec1 := t1, (1, 3)
≤
t2 ≤ t1 ∧ (t2 < t1 ⇒ exec1 ≠ (1, 3)) ∧ inchar ≠ ⊥ ∧ inchar ≠ last →
  msg, last, t2 := msg ∩ inchar, inchar, t2 + chread; (2, 1)
do    t1 ≤ t2 ∧ inchar ≠ ⊥ →
          inchar, t1, exec1 := ⊥, t1 + chsep, (1, 3) (1, 3)
    [] t2 < t1 ∧ (inchar = ⊥ ∨ inchar = last ∨ exec1 = (1, 3)) →
          t2 := t1 (W)
od

```

```

inchar, last, msg :=  $\perp$ ,  $\perp$ ,  $\emptyset$ ;
do-rt    in  $\neq \emptyset \wedge \text{inchar} = \perp \rightarrow$ 
          in, inchar,  $t_1 := \text{tl}(\text{in}), \text{hd}(\text{in}), t_1 + \text{chdef}$ 
           $\square$  inchar  $\neq \perp \rightarrow$  inchar,  $t_1 := \perp, t_1 + \text{chsep}$  (1, 3)
           $\square$  inchar  $\neq \perp \wedge \text{inchar} \neq \text{last}$  without  $\{(1, 3)\} \rightarrow$ 
          msg, last,  $t_2 := \text{msg} \hat{\cap} \text{inchar}, \text{inchar}, t_2 + \text{chread}$  (2, 1)
od-rt

inchar, last, msg,  $t_1, t_2 := \perp, \perp, \emptyset, 0, 0$ ;
do     $t_1 \leq t_2 \wedge \text{in} \neq \emptyset \wedge \text{inchar} = \perp \rightarrow$ 
        in, inchar,  $t_1, \text{exec}_1 := \text{tl}(\text{in}), \text{hd}(\text{in}), t_1 + \text{chdef}, (1, 2)$ 
         $\square$   $t_1 \leq t_2 \wedge \text{inchar} \neq \perp \rightarrow$ 
        inchar,  $t_1, \text{exec}_1 := \perp, t_1 + \text{chsep}, (1, 3)$  (1, 3)
         $\square$   $t_2 \leq t_1 \wedge (t_2 < t_1 \Rightarrow \text{exec}_1 \neq (1, 3))$ 
         $\wedge \text{inchar} \neq \perp \wedge \text{inchar} \neq \text{last} \rightarrow$ 
        msg, last,  $t_2 := \text{msg} \hat{\cap} \text{inchar}, \text{inchar}, t_2 + \text{chread}$  (2, 1)
         $\square$   $t_2 < t_1 \wedge (\text{inchar} = \perp \vee \text{inchar} = \text{last} \vee \text{exec}_1 = (1, 3)) \rightarrow$ 
         $t_2 := t_1$  (W)
od

```

Figure 6 Two processes.

to obtain the action system in Figure 6. In the application of the Movers Theorem, we use the fact that $\text{chread} \leq \text{chdef}$ and four invariants,

- $(t_2 < t_1 \wedge \text{inchar} \neq \perp \wedge \text{inchar} \neq \text{last}) \Rightarrow t_1 = t_2 + \text{chdef}$
(used in the verification of the refinement above);
- $(t_1 \leq t_2 \wedge \text{inchar} \neq \perp) \Rightarrow \text{inchar} = \text{last}$
(used in the verification of condition 2.);
- $t_2 \leq t_1$
(used in the verification of conditions 1.(b) and 3.(d));
- $\text{exec}_1 = (1, 3) \Rightarrow \text{inchar} = \perp$
(used to help prove the other invariants).

Some invariants are stronger than required, but we have found the stronger versions easier to prove.

Note that $\text{mutex}_{2,1}$ does not include the first action even though the first action changes *inchar* and (2, 1) reads it. Intuitively, the first action can be thought of as instantaneously updating *inchar* and then waiting for duration *chdef*, during which time (2, 1) may access *inchar*. The action *W* corresponds to the wait action because of the invariant $t_2 \leq t_1$.

4.3 Final implementation

In the final step, we separate the instantaneous update of *inchar* from waiting for duration *chdef*. For this, we introduce a boolean synchronisation variable *r* and initialise it to *false*. This allows us to add $\neg r$ to the guards of the first and second actions. Now we use the Movers Theorem with

$$\begin{array}{l}
 t_1 \leq t_2 \wedge in \neq \emptyset \wedge inchar = \perp \wedge \neg r \rightarrow \\
 \quad in, inchar, t_1, exec_1 := tl(in), hd(in), t_1 + chdef, (1, 2) \\
 \leq \\
 t_1 \leq t_2 \wedge in \neq \emptyset \wedge inchar = \perp \wedge \neg r \rightarrow \\
 \quad in, inchar, r, exec_1 := tl(in), hd(in), true, (1, 1); \quad (1, 1) \\
 \text{do} \quad t_1 \leq t_2 \wedge inchar \neq \perp \wedge r \rightarrow \\
 \quad r, t_1, exec_1 := false, t_1 + chdef, (1, 2) \quad (1, 2) \\
 \text{od}
 \end{array}$$

to obtain an action system which corresponds to the final implementation in Figure 4. As before, there are no left movers, only right movers. The invariant $t_2 < t_1 \Rightarrow \neg r$ is used in the verification of conditions 1.(b) and 3.(d).

The wait action can be left as it is in Figure 6. A wait action corresponding to that in Figure 4 would require the addition of the disjunct

$$t_1 < t_2 \wedge inchar = \perp \wedge (in = \emptyset \vee r)$$

to the guard and changing the body to $t_1, t_2 := \max(t_1, t_2), \max(t_1, t_2)$. However, this change in the wait action is redundant because of the invariant

$$t_1 < t_2 \Rightarrow (inchar \neq \perp \wedge r \wedge t_2 = t_1 + chread).$$

This invariant also ensures that the assertion

$$!(t_1 < t_2 \Rightarrow (1, 3) \notin mutex_{2,1})$$

for action (1, 3) according to Section 3.6 always skips, taking into account the guard of (1, 3), and therefore may be added to the body of (1, 3) as a valid refinement. (If we had $(1, 1) \in mutex_{2,1}$, then the appropriate assertion for (1, 1) could also be added, for the same reason.)

We do not use Theorem 3 directly in obtaining the final implementation in Figure 4 because the abstract action system for Theorem 3 is not refined by Figure 4 and therefore cannot occur in the development. In the abstract action system for Theorem 3, (1, 1) and (1, 2) may execute alternately without (2, 1) ever occurring, with the result that the final output message *msg* remains empty.

5 DISCUSSION

Related work with similar aims is due to Abadi *et al.* (1994), de Boer *et al.* (1996), Fidge *et al.* (1997) and Broy (1997).

The approach of Abadi *et al.* (1994) uses the Temporal Logical of Actions (TLA) to reason about the real-time properties of systems. Unlike our approach, TLA has a global time variable *now*, which is advanced by a tick action. Fairness constraints are used to execute this tick action and the other actions in parallel, whereas our approach does not require reasoning about fairness. The nonZero condition, which states that *now* is not bounded, is verified in order to help prove liveness properties.

As for our approach, Fidge *et al.* (1997) model concurrent real-time systems with action systems, but use traces for the semantics rather than weakest preconditions. The global variable *now* is incremented periodically, resulting in the uniform advance of time across all processes. The variables up_x and ac_x are similar to our t_i and $mutex_{i,j}$. A compositional Hoare-style proof system for a large class of distributed real-time systems is presented by de Boer *et al.* (1996). The proof system is shown to be sound and relatively complete.

Broy (1997) characterises the behaviour of processes by the histories of output variables relative to input variables. These histories are known as streams, and may be discrete sequences or continuous. The behaviours of processes are combined using parallel composition with feedback, and refinement is defined in terms of simulations between abstract and concrete systems.

The model in this paper is able to handle a wide variety of real-time systems while keeping the introduction of special variables to a minimum, and no assumptions are made about the types of statements. Actions associated with real-time systems such as deadlines and delays can be modelled by assertions and time updates $g \rightarrow t_i := \dots$ respectively.

We believe we have obtained a model of real-time systems which is faithful to the intuition we have about such systems in the real world. While the semantics of the model are complicated, we believe that this reflects the complicated interactions in real-time systems, rather than a deficiency in the model.

In action systems with time, there are as yet few results which provide support for stepwise development. The complexity of the calculations in Section 4 show the desirability of being able to reason on the level of real-time action systems (Figure 2) rather than their underlying standard action systems (Figure 3), and more work is required in this area.

Acknowledgements

I wish to thank Colin Fidge and the anonymous referees for their helpful comments. This work was funded by the Information Technology Division of the Defence Science and Technology Organisation.

REFERENCES

- Abadi M. and Lamport L. (1994) An Old-Fashioned Recipe for Real Time. *ACM Transactions on Programming Languages and Systems*, **16**, 1543–71.
- Back R.J.R. (1989) A Method of Refining Atomicity in Parallel Algorithms, in Odijk E., Rem M. and Syre J.-C. (Eds.) *Proc. Parallel Architectures and Languages Europe (PARLE'89)*. Lecture Notes in Computer Science **366**, Springer-Verlag, New York.
- de Boer F.S., Tej H., de Roever W.-P. and van Hulst M. (1996) Compositionality in Real-Time Shared Variable Concurrency, in Jonsson B. and Parrow J. (Eds.) *Proc. Formal Techniques in Real-Time and Fault-Tolerant Systems (FTRTFT'96)*. Lecture Notes in Computer Science **1135**, Springer-Verlag, New York.
- Boom H.J. (1982) A Weaker Precondition for Loops. *ACM Transactions on Programming Languages and Systems*, **4**, 668–77.
- Broy M. (1997) Refinement of Time, in Bertran M. and Rus T. (Eds.) *Proc. Transformation-Based Reactive System Development (ARTS'97)*, Lecture Notes in Computer Science **1231**, Springer-Verlag, New York.
- Butler M.J. and Morgan C.C. (1995) Action Systems, Unbounded Nondeterminism, and Infinite Traces. *Formal Aspects of Computing*, **7**, 37–53.
- Dijkstra E.W. and Scholten C.S. (1990) *Predicate Calculus and Program Semantics*. Springer-Verlag, New York.
- Fidge C.J. and Wellings A.J. (1997) An Action-Based Formal Model for Concurrent Real-Time Systems. *Formal Aspects of Computing*, **9**, 175–207.
- Hayes I.J. and Utting M. (1997) A Sequential Real-Time Refinement Calculus. *Technical Report 97-33*, Software Verification Research Centre, University of Queensland,
http://svrc.it.uq.edu.au/Publications/Technical_reports_1997.html
- Hesselink W.H. (1995) Safety and Progress of Recursive Procedures. *Formal Aspects of Computing*, **7**, 389–411.
- Morgan C.C. (1994) *Programming from Specifications*, 2nd Edition. Prentice-Hall, New York.
- Morris J.M. (1987) Stepwise Refinement and the Programming Calculus. *Science of Computer Programming*, **9**, 287–306.
- Wabenhorst A.K. (1996) Developing Fairness in Terminating and Reactive Programs. *Technical Report TR-1-96*, Programming Research Group, Oxford University,
<http://www.comlab.ox.ac.uk/oucl/publications/tr/TR-1-96.html>
- Wabenhorst A.K. (1997) A Stepwise Development of the Alternating Bit Protocol. *Technical Report TR-12-97*, Programming Research Group, Oxford University,
<http://www.comlab.ox.ac.uk/oucl/publications/tr/TR-12-97.html>

APPENDIX

Proof of Theorem 3:

We need to ensure that the strengthened guards in the real-time action system do not result in premature termination, and that the wait action does not prevent termination.

Let $\mathcal{B}$ be the real-time action system in Figure 3. We wish to show that $\mathcal{A} \leq \mathcal{B}$, for which we will use the characterisation of weakest preconditions in Theorem 1.

First, we prove a preliminary result. Let $\varphi : seq\mathcal{B} \rightarrow seq\mathcal{A}$ map action sequences in $\mathcal{B}$ to the corresponding action sequences in $\mathcal{A}$, discarding any wait actions. We extend φ to infinite action sequences by continuity. We show that for $s : seq\mathcal{B}$, $\varphi(s) \leq s$ by induction on the construction of s . The result clearly holds for $s = \mathbf{skip}$. For an action A and $P : Pred\mathcal{A}$,

$$\begin{aligned}
 & \varphi(s; A)(P) \\
 \Leftrightarrow & \hspace{20em} \text{property of } \varphi \\
 & (\varphi(s); \varphi(A))(P) \\
 \Rightarrow & \hspace{15em} \text{induction hypothesis} \\
 & (s; \varphi(A))(P) \\
 \Rightarrow & \hspace{20em} \text{see below} \\
 & (s; A)(P)
 \end{aligned}$$

For the last step, there are two cases to consider: if $A = W$, then $\varphi(W)(P) \Leftrightarrow P \Rightarrow W(P)$, whereas if $A = (i, j)$, then $\varphi(i, j)(P) \Rightarrow (i, j)(P)$ because strengthening the guard of an action constitutes a refinement.

To show that $\mathcal{A}(P) \Rightarrow \mathcal{B}(P)$ for $P : Pred\mathcal{A}$, we show that each conjunct for $\mathcal{A}(P)$ in Theorem 1 implies the corresponding conjunct for $\mathcal{B}(P)$. For the first conjunct,

$$\begin{aligned}
 & \forall s : seq\mathcal{A} \ s(P \vee \bigvee_{i=1}^M \bigvee_{j=1}^{N_i} g_{i,j}) \\
 \Rightarrow & \hspace{15em} \text{for } s : seq\mathcal{B}, \varphi(s) : seq\mathcal{A} \\
 & \forall s : seq\mathcal{B} \ \varphi(s)(P \vee \bigvee_{i=1}^M \bigvee_{j=1}^{N_i} g_{i,j}) \\
 \Rightarrow & \hspace{10em} \text{above, since } P \vee \bigvee_{i=1}^M \bigvee_{j=1}^{N_i} g_{i,j} : Pred\mathcal{A} \\
 & \forall s : seq\mathcal{B} \ s(P \vee \bigvee_{i=1}^M \bigvee_{j=1}^{N_i} g_{i,j}) \\
 \Rightarrow & \hspace{15em} \text{monotonicity} \\
 & \forall s : seq\mathcal{B} \ s(P \vee (\bigvee_{i=1}^M t_i \neq T) \vee \bigvee_{i=1}^M \bigvee_{j=1}^{N_i} g_{i,j})
 \end{aligned}$$

which is the desired result by Lemma 1.

For the second conjunct in Theorem 1, we show that for all $\kappa : Ord$, there exists $\kappa' : Ord$ such that $\mathcal{H}_{\kappa, \mathcal{A}}^* \Rightarrow \mathcal{H}_{\kappa', \mathcal{B}}^*$. First suppose that $\mathcal{H}_{0, \mathcal{A}}^*$, and let $s : seq_{\omega} \mathcal{B}$. If s contains (i, j) , say $s = W^n; (i, j); s'$ with $n \geq 0$, then

$$\begin{aligned}
& \mathcal{H}_{0, \mathcal{A}}^* \\
\Leftrightarrow & \text{definition of } \mathcal{H}_{0, \mathcal{A}}^* \\
& \neg \bigvee_{i=1}^M \bigvee_{j=1}^{N_i} g_{i,j} \\
\Rightarrow & W \text{ does not change variables in } \neg \bigvee_{i=1}^M \bigvee_{j=1}^{N_i} g_{i,j} \\
& W^n (\neg \bigvee_{i=1}^M \bigvee_{j=1}^{N_i} g_{i,j}) \\
\Rightarrow & \text{guard of } (i, j) \text{ is } false \text{ if } \neg \bigvee_{i=1}^M \bigvee_{j=1}^{N_i} g_{i,j} \\
& (W^n; (i, j)) ((\bigwedge_{i=1}^M t_i = T) \wedge \neg \bigvee_{i=1}^M \bigvee_{j=1}^{N_i} g_{i,j}) \\
\Rightarrow & \text{definition of } \mathcal{H}_{0, \mathcal{B}}^* \text{ and Lemma 1} \\
& \exists m < \omega \ s|_m (\mathcal{H}_{0, \mathcal{B}}^*)
\end{aligned}$$

On the other hand, if s consists only of wait actions, then execution of at most M of these gives the same result.

For the step case, take $s : seq_{\omega} \mathcal{B}$. If $\varphi(s) : seq_{\omega} \mathcal{A}$, then s has a tail of wait actions, and a sequential composition of at most M of these results in termination, as in the base case. Otherwise, $\varphi(s) : seq_{\omega} \mathcal{A}$ and

$$\begin{aligned}
& \mathcal{H}_{\kappa+1, \mathcal{A}}^* \\
\Leftrightarrow & \text{definition of } \mathcal{H}_{\kappa+1, \mathcal{A}}^* \\
& \forall s' : seq_{\omega} \mathcal{A} \ \exists m < \omega \ s'|_m (\mathcal{H}_{\kappa, \mathcal{A}}^*) \\
\Rightarrow & \varphi(s) : seq_{\omega} \mathcal{A} \\
& \exists m < \omega \ \varphi(s)|_m (\mathcal{H}_{\kappa, \mathcal{A}}^*) \\
\Leftrightarrow & \\
& \exists m < \omega \ \varphi(s|_m) (\mathcal{H}_{\kappa, \mathcal{A}}^*) \\
\Rightarrow & \text{from result earlier in proof} \\
& \exists m < \omega \ s|_m (\mathcal{H}_{\kappa, \mathcal{A}}^*) \\
\Rightarrow & \text{induction hypothesis and monotonicity} \\
& \exists m < \omega \ s|_m (\mathcal{H}_{\kappa', \mathcal{B}}^*)
\end{aligned}$$

As κ' is independent of s , we have shown $\mathcal{H}_{\kappa'+1, \mathcal{B}}^*$.

The limit case is straightforward. □

BIOGRAPHY

Axel Wabenhorst received his doctorate from the University of Oxford in 1997 with a thesis on the formal development of concurrent systems with fairness constraints. His background is in pure mathematics, having completed a science degree at Monash University and Part III Mathematics at the University of Cambridge. Currently, he is a Research Fellow in the Software Verification Research Centre at the University of Queensland, with research interests in the formal development of concurrent systems.