TAM: A Formal Framework for the Development of Distributed Real-Time Systems*

D. J. Scholefield, H.S.M. Zedan
Formal Systems Research Group
Department Of Computer Science
University of York
Heslington, York, YO1 5DD
email: { djs,zedan }@ac.uk.york.minster

Abstract

The Temporal Agent Model (TAM) is a wide-spectrum development language for real-time systems. In TAM, limited resources are modelled by deriving release times and absolute deadlines from weakest pre-condition predicate transformers. In this paper the language syntax and semantics are described along with a number of examples.

1 Introduction

The formal development of a real-time system can be divided into three separate tasks; requirements specification, concrete design, and verification between the former two activities. In real-time systems, verification must take into account not only functional correctness but also the timeliness of results.

In many real-time formal theories, two independent languages are used for the specification and design tasks. For example, in Timed-CSP [Dav91] [Schn90] and Timed-CCS [TM89] first-order predicate logic is used as the requirements specification language, and a process algebra defines the concrete design language. Verification is undertaken by proving that the logic formulae satisfy the traces arising from the processes.

However, a problem inherent in such 'two tiered' theories is the lack of method by which suitable designs are arrived at. A combination of experience and guess-work must be used in order to develop processes, and then verification – a rather time consuming activity – is undertaken. If the verification fails, then the processes are re-formulated and verification is undertaken again. There are some methods which overcome this specific problem, Ostroff [Ost89] has devised a synthesis method which enables the user to derive designs from specifications, however the method is conservative (it does not always find a suitable design even when one may exist). Other methods suffer from similar drawbacks.

To overcome this problem we have devised the Temporal Agent Model (TAM), which is a theory centered upon a 'wide- spectrum' language in which both real-time requirements specifications and concrete designs may be formulated. A specification in TAM may be transformed step-by-step into a mixed program containing both specification statements and concrete design

*Partially supported by UK SERC grant GR/F 35920 /4/1/1214

statements. Such transformations continue until a completely concrete program is produced which is guaranteed correct with respect to the original specification. The transformation process is known widely as refinement, and is already common in purely transformational systems [Mor90] [SETL85] [CIP85].

TAM uses conservative extensions to first-order predicate logic to express the time at which variables hold certain values. The logic is used both as a basis for a language statement, and to define the timed predicate transformers for the language's semantics. The concrete language statements are loosely based on Dijkstra's guarded command language [Dij76] with real-time extensions to cover deadlines, delays and concurrency.

The paper is structured as follows. In the next section the syntax of the TAM language is defined, and an informal semantics is given. In the following section, the semantics are defined and a number of properties of those semantics discussed. A few examples are presented and analysed. The conclusion discusses the definition of a refinement relation, and future work.

2 TAM Syntax

The modular abstraction mechanism in TAM is the *agent* which represents an individually schedulable task with an associated release time and deadline. TAM is a wide-spectrum language in which both specifications and concrete designs may be expressed. In TAM, both specifications and designs are referred to as agents.

2.1 Assumptions

A prime consideration in designing a real-time language is its' amenability to scheduling analysis, therefore in TAM the following assumptions have been made:

- Time is discrete, linear, and global. It is modelled by the natural numbers.
- TAM agents may only be reasoned about if they are guaranteed to terminate within finite time.
- Communication is achieved within TAM agents by reading and writing shared memory spaces called *shunts*. A shunt always contains a single value (resulting from the most recent write), and a time-stamp corresponding to the time of the most recent write. Shunt reading and writing is asynchronous. Shunts may only be written by a single agent, but may be read simultaneously by different agents.
- Computation, such as assignment, input, and output, may not occur instantaneously.

2.2 Agents

The syntax of a TAM agent is defined recursively:

$$\mathcal{A} ::= \quad x := e \quad | \quad [t]\mathcal{A} \quad | \quad g \uparrow \mathcal{A} \quad | \quad \mathcal{A};\ \mathcal{A} \quad | \quad w : \Phi \quad | \quad \mu\, X.\mathcal{A}$$
$$| \quad X \quad | \quad x \rightarrow s \quad | \quad x \leftarrow s \quad | \quad \mathcal{A} \mid \mathcal{A} \quad | \quad \mathcal{A}/s \quad | \quad (x)\mathcal{A} \quad | \quad \delta$$

The agent $x := e$ (where x is some variable name, and e is some expression on variable names) results in the value of e being bound to x. It is assumed that there exists a suitable expression language for e.

The agent $[t]\mathcal{A}$ is guaranteed to terminate within t time units of its release time. The value t must be a constant natural number.

The agent $g \uparrow \mathcal{A}$ (where g is some boolean variable or constant) evaluates g, and if 'true', releases the agent $\mathcal{A}$ immediately. Otherwise no computation takes place and the agent terminates immediately without changing any variables or shunts.

The agent $\mathcal{A};\ \mathcal{A}$ specifies a precendence relation between the agents. The second agent will not be released until the first has terminated.

The specification $w : \Phi$ defines a *frame* w as a list of variable and shunt names whose bindings may change during the execution of an agent whose specification is given as the first-order predicate logic formula Φ. This agent is discussed in more detail in the next section.

The agent $\mu\, X.\mathcal{A}$ is recursive; $\mathcal{A}$ is released, and upon encountering the agent variable X, $\mu\, X.A$ is released again.

The agent $x \rightarrow s$ outputs the value of the variable or constant x into the shunt s with the current time as a time-stamp. The output may occur at any time between the release and termination time of the agent.

The agent $x \leftarrow s$ inputs the time-stamp and value pair from the shunt s into the variable x. The input may occur at any time between the release and termination time of the agent.

The agent $\mathcal{A} \mid \mathcal{A}$ releases the two agents immediately, and terminates some arbitrary time after both agents terminate. The well-formed test on concurrent agent states that no agent may read a variable which may be changed in a concurrently executing agent. The test is defined in detail below

The agent $\mathcal{A}/s$ restricts the shunt s to the agent $\mathcal{A}$. No other agent may read or write to this shunt.

The agent $(x)\mathcal{A}$ restricts the variable x to the agent $\mathcal{A}$. On entry to $\mathcal{A}$ any existing binding to x is replaced by a non-deterministic binding. On termination, the original binding is restored.

The agent δ delays for at least one time unit before terminating, and does not change any variable or shunt binding. The shorthand δn is defined by:

$$\delta 1 =_{def} \delta \qquad \delta n =_{def} \delta;\ \delta n - 1 \qquad (\text{where } n \geq 1)$$

2.3 Specifications

The first-order predicate logic is extended with terms of the form '$x@t$' which represent the value of the computation variable 'x' at time 't'. Such extensions are conservative (they may be defined by functions from natural numbers onto some value set). Shunts are defined similarly, with values sets constructed from pairs of natural numbers and some value set. Shunt values (pairs) may be accessed by the usual 'dot' accessors: $(s@t).1$ (time-stamp) and $(s@t).2$ (value).

The function '*vars*' is defined over first-order predicate logic formulae listing timed variables occurring in the formulae (and similarly, a function '*shunts*'). The functions '$vars^n$' (for natural numbers n) list variables which occur at time n, (similarly '$shunts^n$') (these functions are only well-formed for closed formulae).

In a specification statement $w : \Phi$, w lists those variables and shunts which may change during the execution of an agent which satisfies Φ. The set w is called a frame. The formula Φ may contain the free natural number variables t and t', representing the agent's release and termination time. Computation variables may only be referred to at times t and t', and shunts may only be referred to at times in $[t, t']$. The formula therefore represents a timed transition on computation variables and shunts.

The notation $w.vars$ refers to the computation variables in a frame, and $w.shunts$ refers to the shunts.

2.4 Well-Formed Conditions

Three functions are defined which list those variables or shunts which are changed or read by an agent.

The function $\mathcal{C}$ lists variables which may be changed by an agent:

$$\mathcal{C}(x := e) = \{x\} \quad \mathcal{C}(x \leftarrow s) = \{x\} \quad \mathcal{C}(\delta) = \emptyset \quad \mathcal{C}(x \rightarrow s) = \emptyset$$

$$\mathcal{C}(w : \Phi) = w.vars \quad \mathcal{C}([t]\mathcal{A}) = \mathcal{C}(\mathcal{A}) \quad \mathcal{C}(\mathcal{A} \mid \mathcal{B}) = \mathcal{C}(\mathcal{A}) \cup \mathcal{C}(\mathcal{B})$$

$$\mathcal{C}(\mathcal{A};\ \mathcal{B}) = \mathcal{C}(\mathcal{A}) \cup \mathcal{C}(\mathcal{B}) \quad \mathcal{C}(\mathcal{A}/s) = \mathcal{C}(\mathcal{A}) \quad \mathcal{C}((x)\mathcal{A}) = \mathcal{C}(\mathcal{A}) - x$$

$$\mathcal{C}(g \uparrow \mathcal{A}) = \mathcal{C}(\mathcal{A})$$

The function $\mathcal{R}$ lists those variables which may be read by an agent:

$$\mathcal{R}(x := e) = \mathcal{R}(e) \quad \mathcal{R}(x \leftarrow s) = \emptyset \quad \mathcal{R}(\delta) = \emptyset \quad \mathcal{R}(x \rightarrow s) = \{x\}$$

$$\mathcal{R}(w : \Phi) = vars(\Phi) \quad \mathcal{R}([t]\mathcal{A}) = \mathcal{R}(\mathcal{A}) \quad \mathcal{R}(\mathcal{A} \mid \mathcal{B}) = \mathcal{R}(\mathcal{A}) \cup \mathcal{R}(\mathcal{B})$$

$$\mathcal{R}(\mathcal{A};\ \mathcal{B}) = \mathcal{R}(\mathcal{A}) \cup \mathcal{R}(\mathcal{B}) \quad \mathcal{R}(\mathcal{A}/s) = \mathcal{R}(\mathcal{A}) \quad \mathcal{R}((x)\mathcal{A}) = \mathcal{R}(\mathcal{A}) - x$$

$$\mathcal{R}(g \uparrow \mathcal{A}) = \mathcal{R}(g) \cup \mathcal{R}(\mathcal{A})$$

Note that the function $\mathcal{R}$ is overloaded on expressions and boolean guards.

The function $\mathcal{W}$ lists those shunts that an agent may write to:

$$\mathcal{W}(x := e) = \emptyset \quad \mathcal{W}(x \leftarrow s) = \emptyset \quad \mathcal{W}(\delta) = \emptyset \quad \mathcal{W}(x \rightarrow s) = \{s\}$$

$$\mathcal{W}(w : \Phi) = w.shunts \quad \mathcal{W}([t]\mathcal{A}) = \mathcal{W}(\mathcal{A}) \quad \mathcal{W}(\mathcal{A} \mid \mathcal{B}) = \mathcal{W}(\mathcal{A}) \cup \mathcal{W}(\mathcal{B})$$

$$\mathcal{W}(\mathcal{A};\ \mathcal{B}) = \mathcal{W}(\mathcal{A}) \cup \mathcal{W}(\mathcal{B}) \quad \mathcal{W}(\mathcal{A}/s) = \mathcal{W}(\mathcal{A}) - s \quad \mathcal{W}((x)\mathcal{A}) = \mathcal{W}(\mathcal{A})$$

$$\mathcal{W}(g \uparrow \mathcal{A}) = \mathcal{W}(\mathcal{A})$$

Agents can not read or change variables which may be changed by concurrently executing agents, i.e. in $\mathcal{A} \mid \mathcal{B}$:

$$(\mathcal{C}(\mathcal{A}) \cup \mathcal{R}(\mathcal{A})) \ \cap \ \mathcal{C}(\mathcal{B}) \ = \ \emptyset$$

Agents can not write to shunts that may be written by concurrently executing agents, i.e. in $\mathcal{A} \mid \mathcal{B}$:

$$\mathcal{W}(\mathcal{A}) \cap \mathcal{W}(\mathcal{B}) \ = \ \emptyset$$

2.5 Implications and Examples

A system is described by a single agent which is assumed, without loss of generality, to be released at time 0. For all but the most simple of systems the agent will be concurrent, and all sub-agents will be released at time 0. Later release times are then specified by the use of the δ agent.

For example, consider the following agent:

$$[5](x \leftarrow s;\ \ \delta 3;\ \ x.2 \rightarrow s') \quad \mid \quad [2](0 \rightarrow s'')$$

Which denotes two concurrent agents that are released at time 0. The sub-agents of the left hand agent may be scheduled in any way which respects the precedences defined by the sequence operators and timing constraints (there is only one such schedule as the overall deadline provides five time units, and each of the five sub-agents needs a minimum of one time unit). The right hand agent may perform the write at either time=1 or time=2.

On a single processor, this system requires that the two left hand side communications take a maximum of one time unit each, but the right hand agent write may take up to two time units. A possible schedule (the most relaxed) is therefore[1]:

Agent	execution start time	deadline (absolute)
$x \leftarrow s$	0	1
$0 \rightarrow s''$	1	2
$x.2 \rightarrow s'$	4	5

The following agent is a specification of an operation that finds any index into a boolean array 'a' which corresponds to the position of a 'true' value:

$$\{i\} : \exists\, n \in [1, \#a](a@t[n]) \ \Rightarrow \ a@t[i@t']$$

(where $\#a$ denotes the length of the array). This agent may be specified in a more concrete manner by:

[1]The task of finding a schedule given deadline, release time, and communication information is in general NP-hard, and a sophisticated analysis tool is required. The TAM theory does not aim to provide such schedules, only the information needed to perform the analysis

$$
\begin{array}{l}
(g)(n)(n := 1; \\
\quad \mu X.(g := n > \#a; \\
\quad\quad \neg g \uparrow (\{i, n\} : (a@t[n@t] \;\Rightarrow\; i@t' = n@t) \wedge \\
\quad\quad\quad\quad\quad\quad (\neg a@t[n@t] \;\Rightarrow\; i@t' = i@t) \wedge \\
\quad\quad\quad\quad\quad\quad (n@t' = (n@t) + 1);\; X)) \\
\quad)
\end{array}
$$

Or in a completely concrete agent:

$$
\begin{array}{l}
(g)(n)(n := 1; \\
\quad \mu X.(g := n > \#a;\;\; \neg g \uparrow (g := a[n]; \\
\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad g \uparrow i := n; \\
\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad n := n + 1;\; X)) \quad)
\end{array}
$$

TAM makes no distinction between agents based upon their level of abstraction.

More interesting, are agents of the form:

$$(b \uparrow \delta);\;\; (\neg b \uparrow (\delta;\; \delta))$$

where timing information is determined at run time as a result of computation. This is a very flexible and powerful part of the TAM language, but care must be taken to ensure that schedulability is not compromised. In the case of variable delays the problem is one of efficiency (the worst-case must be assumed), but with agents of the form:

$(b \uparrow [n]\mathcal{A});\;\; (\neg b \uparrow [m]\mathcal{B})$ (where n,m are constants)

the problem is much more complex. In general, dynamically changing deadlines prove difficult to guarantee, and the use of the TAM language may have to be restricted in order to ensure that programs are amenable to schedulability analysis.

Consider the following agent:

$$\mathcal{A}/s \;\mid\; \mathcal{B}/s$$

The act of restricting a shunt is the same as renaming within the agent being restricted (along with some semantic transformations which we will not consider here). Therefore, this agent is exactly equivalent to:

$$\mathcal{A}[s'/s]/s' \;\mid\; \mathcal{B}[s''/s]/s''$$

The question of when the shunt or variable is restricted is therefore not relevant, only for which agent it is restricted for.

Consider the agent:

$$\prod_{n \in [0,t]} [n]\delta n$$

where '$\prod$' represents indexed concurrency and t is some very large natural number (i.e. distant time). This agent also uses no resources, but executes for large amount of time (until the time t) and eventually terminates. This agent is perfectly well-behaved, and may be reasoned about.

Consider the agent:

$$\text{false} \uparrow x := x \quad | \quad x := y + 1$$

which performs assignments to the variable x in parallel. This agent is ill-formed as it refers to the variable x in both parts of the concurrent agent. Even though one of the agents will never actually use x, it remains ill-formed (the well-formed test is pessimistic).

The following agent is not ill-formed:

$$y := x \quad | \quad z := x$$

(both agents refer to x but they are both reads.)

3 TAM Semantics

The semantics of TAM are described by predicate transformers over agents, execution intervals, post-conditions (which describe the functional and timing requirements of an agent), and communication side-effects (which describe the outputs caused by an agent).

3.1 Post-Conditions, Pre-Conditions and Environments

A post-condition is a first-order predicate logic formula over the value of computation variables at the termination time of an agent, and shunt values at any times. The post-condition therefore represents the user's requirements, and the time at which those requirements must be met.

For example, the post-condition:

$$x@5 \geq 0$$

requires that upon termination of an agent (which must be before or at time=5) the value of the computation variable x must be greater than, or equal to, zero. The post-condition:

$$s@8.1 \geq 0$$

asserts that the shunt s must be written to with some value before time=8 (either by the agent itself, or by some agent which is executing concurrently).

Similarly, a pre-condition is a first-order predicate logic formula over the value of computation variables at the release time of an agent, and shunt values at any time, along with a communication side-effect. The pre-condition represents the required initial state, and communications that are needed in order to guarantee some post-condition holds at the termination of an agent.

A communication side-effect is a first-order predicate logic formula which describes the outputs caused by an agent. It is used to discharge input requirements of concurrently executing agents.

Every agent has a corresponding environment which is a set of shunt names listing those shunts which may be changed by the agent.

3.2 Predicate Transformers

A predicate transformer describes the way in which a post-condition and a communication side-effect are syntactically transformed into the weakest pre-condition that needs to hold in order to guarantee the post-condition.

A transformer is a function over pairs of first-order predicate logic formulae, a release time, a termination time, an agent, and an environment, i.e:

$$wp_\delta^u(\mathcal{A},\ \phi, \psi) : w$$

where 'δ' is the relase time, 'u' is the termination time ($\delta \leq u$), '$\mathcal{A}$' is some agent, 'ϕ' is the post-condition, 'ψ' is the side-effect, and 'w' is an environment. The result is a pre-condition formula and a side-effect formula.

The functional part of the pair of formulae is referred to by $(\phi, \psi).f$, and the communication side-effect as $(\phi, \psi).c$, i.e:

$$(\phi, \psi) \ \equiv \ ((\phi, \psi).f,\ (\phi, \psi).c)$$

3.3 Supporting Concurrency

The following predicates appear commonly in the transformer rules.

The predicate $stable(s, \delta, u)$ asserts that the shunt s does not change value within the interval $[\delta, u]$:

$$stable(s, \delta, u) \ =_{def} \ \bigwedge_{\sigma \in [\delta, u]} s@\sigma = s@(\sigma - 1)$$

(where $0 - 1 = 0$). The predicate *stable* is extended over sets:

$$all_stable(S, \delta, u) \ =_{def} \ \bigwedge_{s \in S} stable(s, \delta, u)$$

The predicate $write(v, s, \delta, u)$ asserts that the value v is written to the shunt s sometime in the interval $[\delta + 1, u]$. The shunt remains stable at other times:

$$write(v, s, \delta, u) \ =_{def}$$

$$\bigvee_{\sigma \in [\delta+1, u]} (s@\sigma = (\sigma, v) \ \wedge \ stable(s, \delta, \sigma - 1) \ \wedge \ stable(s, \sigma + 1, u))$$

3.4 Transformer Definitions

In the following predicate transformers, the notation $[\![\phi]\!]_\delta^u$ represents the simultaneous replacement of all computation variables at time u in ϕ with the same variables at time δ (but *not* shunts), e.g:

$$[\![x@u = (y@\delta) + 1 \ \wedge \ s@u = (u, y@u) \ ]\!]_\delta^u \ \equiv \ \ x@\delta = (y@\delta) + 1 \ \wedge \ s@u = (u, y@\delta)$$

(where s is a shunt)

3.4.1 Assignment

$$wp_\delta^u(x := e, \phi, \psi) : w \ =$$

$$(\delta < u) \ \wedge \ [\![\phi[e@\delta / x@u]]\!]_\delta^u,\ (\delta < u) \ \wedge \ [\![\psi[e@\delta / x@u]]\!]_\delta^u \ \wedge \ all_stable(w, \delta, u)$$

The expression e is evaluated with the bindings found at time δ. The result is bound to x at or before time u. The evaluation will take at least one time unit, and no shunts in the agent's environment are changed.

3.4.2 Deadline

$$wp^u_\delta([t]\mathcal{A}, \phi, \psi) : w \;=\; wp^n_\delta(\mathcal{A}, [\![\phi]\!]^u_n, [\![\psi]\!]^u_n) : w \qquad (\text{where } n = \text{minimum}(\delta + t, u))$$

The deadline declaration shortens the execution window only if the time given (t) is less than that already available in $[\delta, u]$.

3.4.3 Guard

$$wp^u_\delta(g \uparrow \mathcal{A}, \phi, \psi) : w \;=$$

$$(g@\delta \Rightarrow wp^u_\delta(\mathcal{A}, \phi, \psi) : w.f) \;\wedge\; (\neg g@\delta \Rightarrow [\![\phi]\!]^u_\delta),$$

$$(g@\delta \Rightarrow wp^u_\delta(\mathcal{A}, \phi, \psi) : w.c) \;\wedge\; (\neg g@\delta \Rightarrow [\![\psi]\!]^u_\delta \;\wedge\; all_stable(w, \delta, u))$$

The guard is a boolean variable and therefore takes no time to evaluate. If it is true, then the execution of the guarded agent takes place immediately. If not, then the guarded agent terminates immediately, and no variables or shunt values are changed.

3.4.4 Sequential

$$wp^u_\delta(\mathcal{A};\ \mathcal{B}, \phi, \psi) : w \;=$$

$$\bigvee_{\sigma \in [\delta, u]} wp^\sigma_\delta(\mathcal{A},\, wp^u_\sigma(\mathcal{B}, \phi, \psi) : w) : w.f, \qquad \bigvee_{\sigma \in [\delta, u]} wp^\sigma_\delta(\mathcal{A},\, wp^u_\sigma(\mathcal{B}, \phi, \psi) : w) : w.c$$

The sequential agent requires that there is at least one intermediate time in the execution window $[\delta, u]$ before which $\mathcal{A}$ will terminate and after which $\mathcal{B}$ is released successfully.

3.4.5 Specification

$$wp^u_\delta(w : \Phi, \phi, \psi) : w' \;=$$

$$\#\Phi + \delta \leq u \;\wedge\; [\![\forall\, w.vars@u(\exists\, w.shunts(\Phi[\delta/t][u/t']) \Rightarrow \phi)]\!]^u_\delta,$$

$$\exists (w.vars - vars^u(\psi))@u(\Phi[\delta/t][u/t']) \;\wedge\; \psi[y@\delta / y@u]_{(\text{all } y \text{ not in } w)}$$

$$\wedge\; stable(w' - w, \delta, u)$$

The quantifiers $\forall$ and $\exists$ are extended over computation variables and shunts by the definition:

$$\forall\, x@t(\phi) \;=_{def}\; \forall\, x(\phi[x/x@t])$$

$$\exists\, x@t(\phi) \;=_{def}\; \exists\, x(\phi[x/x@t])$$

These definitions are extended over sets of variables and shunts in the usual way. Variables and shunts which are quantified without specific times are assumed to be quantified for all times at which they occur in the formula.

The notation $\#\Phi$ denotes the *degree* of the specification Φ, which is the least amount of time which is needed to guarantee that the specification is feasible (logically consistent) for any execution interval. In order to define the degree of a specification we first need to define timing-feasibility.

A formula Φ, with natural number variables t and t' occurring free (but which is otherwise closed) is timing-feasible under n if n is the smallest natural number such that for every interval

$[\delta, u]$ there exists an interpretation I such that:

$$I \models (\delta + n \leq u) \Rightarrow \Phi[\delta/t][u/t']$$

The degree of a specification is therefore the specification's timing-feasibility number.

3.4.6 Recursion

$$wp^u_\delta(\mu X.\mathcal{A}, \phi, \psi) : w =$$

$$\bigvee_{n \in \mathcal{N}} wp^u_\delta(\mathcal{A}^n(\text{abort}), \phi, \psi) : w.f, \quad \bigvee_{n \in \mathcal{N}} wp^u_\delta(\mathcal{A}^n(\text{abort}), \phi, \psi) : w.c$$

Where $\mathcal{A}^0(\mathcal{B}) =_{def} \mathcal{B} \qquad \mathcal{A}^{n+1}(\mathcal{B}) =_{def} \mathcal{A}[\mathcal{A}^n(\mathcal{B})/X]$

(if X is the recursion variable).

Where $\text{abort} =_{def} [0]\delta$

3.4.7 Output

$$wp^u_\delta(x \rightarrow s, \phi, \psi) : w =$$

$$\delta < u \ \wedge \ [\![\phi]\!]^u_\delta, \ write(x@\delta, s, \delta, u) \ \wedge \ all_stable(w - s, \delta, u)$$

The output agent requires at least one time unit in which to execute (this condition is implicitly asserted in the communication side-effect because an empty disjunction in the 'write' predicate gives rise to the predicate 'false').

3.4.8 Input

$$wp^u_\delta(x \leftarrow s, \phi, \psi) : w =$$

$$\delta < u \ \wedge \ [\![\bigwedge_{\sigma \in [\delta+1, u]} \phi[s@\sigma/x@u]]\!]^u_\delta, \qquad \delta < u \ \wedge \ all_stable(w, \delta, u)$$

The input requires at least one time unit in which to execute, and asserts that all of the values held in the shunt s in the appropriate interval must fulfill the pre-condition. The shunts remain unchanged during an input.

3.4.9 Concurrency

$$wp^u_\delta(\mathcal{A} \mid \mathcal{B}, \phi, \psi) : w =$$

$$\exists X^*(wp^u_\delta(\mathcal{A}, \phi_\mathcal{A}, \psi_\mathcal{A}) : w - \mathcal{W}(\mathcal{B}).f \ \wedge \ wp^u_\delta(\mathcal{B}, \phi_\mathcal{B}, \psi_\mathcal{B}) : w - \mathcal{W}(\mathcal{A}).f),$$

$$\exists Y^*(wp^u_\delta(\mathcal{A}, \phi_\mathcal{A}, \psi_\mathcal{A}) : w - \mathcal{W}(\mathcal{B}).c \ \wedge \ wp^u_\delta(\mathcal{B}, \phi_\mathcal{B}, \psi_\mathcal{B}) : w - \mathcal{W}(\mathcal{A}).c)$$

The division of the functional predicate and communication predicate is dictated by the variables occurring in each agent. In both cases, the conjunction of the two predicates, along with the quantified variables found in the two sets X^* and Y^* must be equivalent to the original. In addition, the predicates may only refer to variables not changed by concurrently executing agents.

3.4.10 Restriction

$$wp^u_\delta(\mathcal{A}/s, \phi, \psi) : w \;=$$

$$\text{discharge}(wp^u_\delta(\mathcal{A}, \phi, \psi) : w) \;\wedge\; \neg(s \in \mathit{shunts}(\text{discharge}(wp^u_\delta(\mathcal{A}, \phi, \psi) : w))),$$

$$\exists\, s(wp^u_\delta(\mathcal{A}, \phi, \psi) : w.c)$$

The restriction of a shunt removes any shunt writes from the environment, and discharges any obligations arising in the functional pre-condition on that shunt. If the shunt occurs within the post-condition or side-effect, then it is renamed to a unique shunt name within the agent.

The discharge function is defined over predicates thus:

$$\text{discharge}(\phi, \phi') \;= \text{the weakest formula such that:}$$

$$\text{discharge}(\phi, \phi') \;\wedge\; \phi' \quad \equiv\rangle \quad \phi.$$

3.4.11 Local Variable

$$wp^u_\delta((x)\mathcal{A}, \phi, \psi) : w \;=$$

$$\exists\, x@\delta(wp^u_\delta(\mathcal{A}, \phi, \psi) : w.f), \qquad \exists\, x@\delta(wp^u_\delta(\mathcal{A}, \phi, \psi) : w.c)$$

The restriction of a local variable removes any changes to that variable from the functional and communication predicates. If the variable occurs in the post-condition or side-effect, then it is renamed to a unique variable within the agent.

3.4.12 Delay

$$wp^u_\delta(\delta, \phi, \psi) : w \;=$$

$$\delta < u \;\wedge\; [\![\phi]\!]^u_\delta,\; \delta < u \;\wedge\; [\![\psi]\!]^u_\delta$$

The delay agent requires at least one time unit in which to execute, and does not change variables or shunts.

3.5 Examples

Consider the simple agent $x := x+1$ executing in a window $[3, 8]$, with an empty environment, then:

$$wp^8_3(x := x + 1, x@8 \geq 1, \text{true}) : \emptyset \quad = \quad x@3 \geq 0, \text{true}$$

i.e. if the value of x at time=3 is at least zero, then incrementing x in the interval $[3, 8]$ will guarantee that the value of x at time=8 is at least one. The transformation is as follows:

$$wp^8_3(x := x + 1, x@8 \geq 1, \text{true}) : \emptyset$$

$$\equiv \quad 3 < 8 \;\wedge\; [\![x@8 \geq 1[(x@3) + 1/x@8]]\!]^8_3,\; 3 < 8 \;\wedge\; \mathit{all_stable}(\emptyset, 3, 8)$$

$$\equiv \quad [\![(x@3) + 1 \geq 1]\!]^8_3, \quad \mathit{all_stable}(\emptyset, 3, 8)$$

$$\equiv \quad [\![x@3 \geq 0]\!]^8_3, \quad \text{true}$$

$$\equiv \quad x@3 \geq 0, \quad \text{true}$$

Consider the agent:

$$\{x\} : x@t' = f(y@t)$$

which assigns the value of $f(y)$ at the release time, to the variable x at the termination time. This agent is not equivalent to the agent:

$$x := f(y)$$

The specification agent may take no time in which to execute (it has a degree of zero). In particular, in the case when $f(y@t) = x@t$, then no action need be taken. In the assignment agent the evaluation will always take at least one time unit.

Consider the agent:

$$[1](x \rightarrow s \mid y \leftarrow s)/s$$

The input and output to shunt s is constrained by the deadline to occur at release-time+1. This guarantees the assignment of the value in x to $y.2$ and the time of the transfer to $y.1$. The following transformation demonstrates the assignment within an execution window of $[0, 5]$:

$$wp_0^5([1](x \rightarrow s \mid y \leftarrow s)/s, y@5.2 = x@0, \text{true}) : \emptyset$$

$$\equiv \quad wp_0^1((x \rightarrow s \mid y \leftarrow s)/s, y@1.2 = x@0, \text{true}) : \emptyset$$

The pre-discharge formulae is calculated first:

$$\equiv \quad \exists\, v(wp_0^1(x \rightarrow s, v = x@0, \text{true}) : \emptyset.f \ \wedge \ wp_0^1(y \leftarrow s, y@1.2 = v, \text{true}) : \emptyset.f)$$

$$\wedge \ \exists\, z(wp_0^1(x \rightarrow s, z = x@0, \text{true}) : \emptyset.c \ \wedge \ wp_0^1(y \leftarrow s, z = y@1.2, \text{true}) : \emptyset.c)$$

Individually:

$$wp_0^1(x \rightarrow s, v = x@0, \text{true}) : \emptyset.f \ \equiv \ v = x@0$$

$$wp_0^1(x \rightarrow s, v = x@0, \text{true}) : \emptyset.c \ \equiv \ s@1 = (1, x@0)$$

$$wp_0^1(y \leftarrow s, v = y@1.2, \text{true}) : \emptyset.f \ \equiv \ s@1.2 = v$$

$$wp_0^1(y \leftarrow s, v = y@1.2, \text{true}) : \emptyset.c \ \equiv \ \text{true}$$

Back in conjunction gives:

$$\exists\, v(v = x@0 \ \wedge \ s@1.2 = v), \ \exists\, z(s@1 = (1, x@0))$$

$$\equiv \quad s@1.2 = x@0, s@1 = (1, x@0)$$

The discharge on the shunt 's' results in:

$$\text{discharge}(s@1.2 = x@0, s@1 = (1, x@0)) \ \equiv \ true$$

$$(\text{because } s@1 = (1, x@0) \equiv\rangle s@1.2 = x@0)$$

Giving the overall pre-condition:

$$\equiv \quad \text{true}, \exists\, s(s@1 = (1, x@0))$$

$\equiv$ true, true

3.6 Properties of The Predicate Transformers

We are primarily interested in two properties: distribution of conjunction and distribution of entailment. Both properties are used in the proof of monotonicity of a refinement relation defined later. The proofs of the following two theorems are available from the authors.

3.6.1 Distribution of Conjunction

Theorem 1: $wp^u_\delta(\mathcal{A},\ \phi \wedge \phi', \psi) : w \ \equiv \ wp^u_\delta(\mathcal{A},\ \phi, \psi) : w \ \wedge \ wp^u_\delta(\mathcal{A},\ \phi', \psi) : w$

Where $(\phi, \psi) \ \wedge \ (\phi', \psi') \ =_{def} \ (\phi \wedge \phi', \psi \wedge \psi')$

3.6.2 Implication Distribution (Monotonicity)

Theorem 2: $\phi \ \equiv\rangle \ \phi' \ \Rightarrow \ wp^u_\delta(\mathcal{A},\ \phi, \psi) : w \ \equiv\rangle \ wp^u_\delta(\mathcal{A},\ \phi', \psi) : w$

Where $(\phi, \psi) \ \equiv\rangle \ (\phi', \psi') \ =_{def} \ (\phi \equiv\rangle \phi') \ \wedge \ (\psi \equiv\rangle \psi')$

4 Practical Real-Time Issues in TAM

In designing a formalism for real-time systems, a major objective is to provide a practical framework in which common real-time issues may be easily expressed. Such issues include the ability to read a global clock, specify periodic tasks, and provide time-outs on communication events. In the following examples the notation:

$\odot_n(\mathcal{A})$ (where $n \geq 1$)

represents the iteration operator defined by:

$\odot_1(\mathcal{A}) =_{def} \mathcal{A}$ $\odot_{n+1}(\mathcal{A}) =_{def} \mathcal{A}; \odot_n(\mathcal{A})$

4.1 Reading The Global Clock

TAM allows access to the global real-time clock in the time-stamps of the shunts. An agent called *Clock* which updates the time-stamp of a boolean shunt called 'time' on each clock-tick is easily derived. All TAM agents must execute for a finite time, and so an arbitrary duration for the clock agent of 'n' time units is used:

$Clock =_{def} [n](\odot_n(\text{true} \rightarrow \text{time}))$

Any agent requiring the current time may read the time-stamp of the shunt 'time'. An alternative solution is for the agent which requires the time to write and read a private shunt:

$(true \rightarrow time;\ \ x \leftarrow time)/time$

However, the latter solution is more time consuming (it takes a minimum of two time units in which to execute, rather that the one time unit it would take to read the Clock shunt). The Clock agent requires resources continuously, but it may be possible to map this simple agent onto a hardware device.

4.2 Periodic Tasks

A periodic task should be released once in every given period. In the following example, an agent form is defined which is parameterised over an agent (which represents the task), a period 'p', a deadline 'd', and a maximum number of invocations 'n':

$$Period(\mathcal{A}, p, d, n) \;=\; [n \times p](\textcircled{:}_{\,n}([d]\mathcal{A} \mid \delta p))$$

The deadline $[n \times p]$ forces each release of $\mathcal{A}$ to occur every p time units. In this example it must also be the case that deadline $\leq$ period.

4.3 Time-Outs

A time-out defines the maximum length of time for which an agent is prepared to wait for an event before it decides on an alternative course of action. An 'event' is defined as the writing of the value 'true' into a boolean shunt. A particularly simple timeout agent form for shunt s, positive agent $\mathcal{A}$ (the agent which is executed if the event occurs on time), negative agent $\mathcal{B}$ (which is executed if the event does not occur on time), and time t is defined as:

$$Timeout(s, t, \mathcal{A}, \mathcal{B}) \;=_{def}\; [t](\delta(t-1);\; x \leftarrow s);\;\; (x.2 \uparrow \mathcal{A});\; (\neg x.2 \uparrow \mathcal{B})$$

This timeout agent is inefficient; if the shunt is written to in the first time-unit after the timeout's release then the positive agent will still not be released until some time after the timeout period. More efficient timeouts can certainly be imagined.

5 Conclusion

In this paper the Temporal Agent Model (TAM) was introduced, a formal framework within which a distributed, reliable, real-time system may be specified and reasoned about in a compositional fashion.

TAM may be seen as a wide-spectrum language which is capable of expressing a real-time system at a range of levels of abstraction within a unified syntax. The abstract primitives enable the user to specify the logical and temporal behaviour of a program without recourse to developing a particular algorithm. The concrete primitives provide a typical imperative programming language based on assignment and flow control structures. The abstract and concrete primitives may be arbitrarily intermixed to provide an extremely flexible syntax in which programs can be expressed in a manner which suits the user's chosen development method.

Wide-spectrum languages are particularly amenable to refinement methods. Program refinement is a process of transformation which increases a particular metricated property, but which respects functionality. Usually the metricated property is 'abstraction', and refinement is used as a generator of implementations from specifications.

The transformation process often involves two separate considerations, operational refinement, which transforms the algorithmic parts of the program, and data refinement which transforms the representation of data structures.

In process algebra, refinement is sometimes seen as an exercise in process simulation [Mil89] [Tof90]. Data refinement is not carried out because most process algebras do not have data structures (they are purely abstract languages). However, both data and operational refinement

is considered in Morgan's refinement calculus, which combines first-order predicate logic specifications over state transitions within Dijkstra's guarded command language [MRG88] [Mor90]. Refinement laws are derived which preserve functionality and reduce abstraction. Furthermore, Back and Wright [BW90] have automated proof obligations in a refinement calculus by implementing a guarded command language's semantics using the Higher Order Logic interpreter (HOL).

Nevertheless, there is little reference to refinement for real-time systems in the literature. It is clear that refinement in real-time systems should also preserve temporal properties along with the functional ones, and in the remainder of this section we investigate how this might be achieved.

Delays provide information to the scheduler about an agent's resource requirements (i.e. that no resources are needed for a specific amount of time). It would initially seem intuitive that a refinement of a TAM agent which delays for some time would be the same agent which delays for a longer period:

$$(\delta t;\ \mathcal{A} \ \sqsubseteq\ \delta t + t';\ \mathcal{A})$$

where '$\sqsubseteq$' represents refinement (read as 'is refined by').

However, there are two problems with this. Firstly, the agent A might require the use of a resource at the specific time provided by t (an input occurs before a shunt is overwritten for example), and in the refined agent this resource would not be available at that time. Secondly, it would mean that the refinement relation was not monotonic, a result which would preclude compositional refinement.

It may also seem intuitive to shorten deadlines. Again, this gives us problems with resource management. If an agent is given a deadline, then it is possible for that agent to execute until the end of that deadline and still meet the specification. If an agent is relying on an input which is only possible at the instant before a deadline, then the agent needs all available execution time in order to meet the specification – shortening that deadline will mean that the specification is inconsistent.

Therefore, refinement respects timing information *absolutely* and only functional requirements may be weakened. A refinement relation is defined as follows:

$$\mathcal{A} \sqsubseteq \mathcal{B} \text{ iff } \quad \forall \delta, u, \ \forall \phi, \psi \ \forall w$$

$$wp^u_\delta(\mathcal{A},\ \phi, \psi) : w.f \ \equiv\rangle \ wp^u_\delta(\mathcal{B},\ \phi, \psi) : w.f$$

$$\wedge \ \ wp^u_\delta(\mathcal{A},\ \phi, \psi) : w.c \ \equiv \ wp^u_\delta(\mathcal{B},\ \phi, \psi) : w.c$$

At present a refinement calculus is being developed which will enable the user to generate concrete code (without specifications) by using a set of syntactic substitution rules. These rules will form an equational theory which is sound with respect to the wp^u_δ semantics.

6 Acknowledgement

The authors wish to acknowledge fruitful discussion with colleagues in the 'Real-Time Systems: The Next Generation' project team at the University of York, and especially Ken Tindell and Clive Adams.

7 References

[BW90] R. Back, J. Wright. "Refinement Concepts Formalised in Higher Order Logic". BCS-FACS Vol 2. No. 3. 1990.

[Dij76] E. Dijkstra. "A Discipline Of Programming". Prentice-Hall. 1976.

[MRG88] C. Morgan, K. Robinson, P. Gardiner. "On the refinement calculus". Technical Report PRG-70. Programming Research Group. 1988

[Mil89] R. Milner. "Communication and Concurrency". Prentice-Hall. 1989.

[Ost89] J. Ostroff. "Temporal Logic for Real-Time Systems". Research Studies Press. 1989.

[Mor90] C. Morgan. "Programming from Specifications". Prentice-Hall. 1990.

[Tof90] C. Tofts. "Timed Concurrent Processes". BCS-Leicester Workshop in Semantics for Concurrency. Springer-Verlag 1990.

[CIP85] The CIP Language Group, "The Munich Project CIP. Vol1", LNCS-183. 1985.

[Dav91] J. Davies, "Specification and Proof in Real-Time Systems". Oxford University, Ph.D Thesis PRG-93. 1991.

[Har88] D. Harrison, "RUTH: A Real-Time Applicative Language", University of Stirling Ph.D. 1988.

[Hoo91] J. Hooman, "Specification and Compositional Verification of Real-Time Systems". Ph.D. Thesis, Eindhoven University. 1991.

[Schn90] S. Schneider, "Correctness and Communication in Real-Time Systems". Oxford University, Ph.D. Thesis PRG-84. 1990.

[SETL85] E. Schonberg, D. Shields, "From Prototype To Efficient Implementation: a Case Study Using SETL and C". Courant institute of mathematical sciences, Dept of computer science, New York University. 1985.

[TM89] C. Tofts, F. Moller. "A Temporal Calculus Of Communicating Systems". Edinburgh University, Technical Report LFCS-89-104.