A Standard for Finite TAM[1]

David Scholefield, Hussein Zedan
Formal Systems Research Group
Department of Computer Science
University of York
Heslington, York, YO1 5DD

August 12, 1993

Abstract

Since its inception in 1991, the Temporal Agent Model theory (TAM) has undergone a number of changes; some of these changes have been minor (such as a change in the style of semantic presentation) and some have been more major (such as the addition or subtraction of programming constructs from the TAM language). Throughout these changes however, the computational model underlying the theory has remained constant, allowing for a comparison of different flavours of the language. This has proved a fruitful area of research, but has also meant that references to the TAM theory may be ambiguous. In this paper we present a standard TAM theory, including a language syntax, semantics, and refinement calculus, and propose that this standard be used in future TAM projects. We also discuss refinement hueristics for ensuring implementable programs.

1 Introduction

In January 1992 the first version of TAM was presented at the Nijmegen workshop on the formal development of real-time and fault tolerant systems [8]. The semantics of the wide-spectrum refinement language were described using extended weakest preconditions (which were later presented in [9]), and the underlying logic was based upon conservative extensions to first-order predicate logic. This version of TAM had single untimed guards, full recursion, but no refinement calculus (although a refinement relation was presented). The main problems with this language stemmed from the lack of any construct with which to define agents which were released as a result of an event occurring (enabling the user to model sporadic real-time tasks), and the complexity involved with trying to find a refinement law for the recursive agent operator. Also, the guards were untimed, which we believed was unrealistic: it led to the execution constraint that guard evaluation had to be instantaneous[2].

In 1992 a version of the TAM language, along with a full refinement calculus, was presented [6]. This version of TAM used the same underlying logic and weakest precondition style of the previous version, but it contained a number of important differences. The guards were given an evaluation period, recursion was replaced with bounded iteration, the deadline and sequence operators were coalesced into a single operator, and a timeout operator was added (which modelled the release of an agent in response to an event). This version of the language also appeared in the Fourth Euromicro Workshop on Real-Time Systems along with a small case study [7]. At this stage, although the language

[1]This work is supported by SERC grant GR/H39611

[2]We discuss this issue in some detail with respect to refinement hueristics.

had become more realistic, the weakest precondition style for describing the semantics had now become too complex to read; this was due to the addition of increasingly powerful language constructs.

As a result, the semantics were finally simplified. A Specificational style semantics were defined using the TAM logic formulae as a direct definition for the language. This resulted in a clarification of both refinement proofs, and the proofs of soundness of refinement laws. This version of the language was presented along with a number of case studies in [10] [11] and [12]. The language presented in these papers had not otherwise changed beyond a few minor syntactic improvements (the use of a single operator for hinding both variables and shunts for example).

Because there are a number of versions of the TAM theory in the literature, in this paper we present a version of TAM which we will be adopting as a standard. This version can then be referenced as *the* TAM, with other publications being explicitly referenced. The language we present is close to those presented in the papers promoting the specificational style semantics. Those versions were perhaps the best compromise between the languages' expressive power and formal complexity.

The remainder of this paper is organised as follows. In the next section we discuss the TAM philosophy, including the underlying computational model of TAM, our view of the limited resources issues, and an argument in support of bounded computation in real-time systems. We then define the TAM real-time logic, the TAM syntax, and TAM semantics. The refinement calculus is then presented, and we discuss a number of hueristics which have been defined in order to guide the refinement process to programs which are executable.

2 The TAM Philosophy

TAM aims to be a *realistic* software development method for real-time systems. It has striven to support a computational model which is amenable both to analysis by run-time execution environment software, and to efficient implementation. In doing so, TAM has not shared any of the simplifying assumptions that other techniques promote, e.g. the maximum parallelism hypothesis (there exist an infinite number of resources available to the program) [4], and the instantaneous communication assumption promoted by many real-time process algebras [1] [5].

The trade-off is that TAM can often appear complex, both in the syntax it provides for specifications, and in the discharging of proof obligations during the verification process. Thus the learning curve for TAM is very steep, but we believe that the eventual pay-off is worth the extra effort: the TAM language not only provides a method for verifying real-time and functional correctness of programs, but also provides a language of great flexibility for discussing general issues in real-time system design. This latter point has been demonstrated in publications by researchers in fields which are not mainstream real-time (for example see [3]).

The TAM theory has also been designed to support a specific development method. Many so-called formal methods only consist of a notation, and not a method which enables the user to carry out a specific list of steps in order to arrive at a correct implementation. The TAM method can be summarised as follows:

- Step 1 – the user describes timing and functional requirements in a specification language based upon simple extensions to first-order predicate logic. The specification also defines the interface between the system and the environment.

- Step 2 – the TAM theory provides a set of laws which enables the user to gradually replace parts of the specification with executable code which is guaranteed to be correct with respect to the specification. This process is known as *step-wise refinement*.

- Step 3 – eventually only executable code remains, and this is analysed by schedulability and allocation tools, compiled, and then executed.

The executable language provides real-time syntactic constructs such as deadlines and timeouts, as well as more conventional constructs such as assignment, loops, concurrent composition, communication and conditionals. There is no provision of any syntax to describe the behaviour of the resources used when the program is executed e.g. there is no syntax to describe which processor each concurrent agent should execute on. This is because we believe that the run-time execution support tools such as schedulers and task allocators, and not the programmer, should provide information such as the placement of agents: programs should be independent of such concerns. If this were not the case then the verification process would have to deal with much more complex issues such as scheduling correctness and task placement, and this would prove infeasible in any realistic computational model.

The TAM software development method therefore results in a number of concurrent agents, which are descriptions of *tasks*, and which include information such as deadlines, delays, precedence constraints etc. We then expect the run-time execution environment tools to place those agents, and decide upon their release times (within the bounds defined by the timing information inherent in the syntactic description of the agents) so as to make the schedulability test succeed. This approach makes the verification manageable, but has the drawback that the scheduler may not be able to find a schedule for the given set of agents, and the development process may be forced to backtrack to ensure that a different set of agents is produced. We are currently investigating ways in which tools such as the scheduler can produce guidance to the refinement process at a very early stage so as to avoid backtracking.

The TAM theory also aims to provide a language which supports the ways in which software engineers already produce software, rather than forcing them to change development practices to suit a particular formal approach. For this reason we provide a *wide-spectrum* language which has a syntax that supports both conventional real-time programming constructs (a kind of Pascal with deadlines, delays, timeouts, and timestamps), and a specification construct in which requirements may be written. The specification construct forms a normal part of the language and may be freely intermixed with other code. A program may then contain assertions on what the programmer requires, as well as algorithms which describe how requirements are going to be met. This language directly supports the step-wise refinement process as well as allowing for a less strict method in which some parts of systems can be specified and refined, and some parts can be written directly in code.

The TAM theory views a real-time system as a set of concurrently executing agents, each with deadline, release offsets, and period or release event. Agents communicate via shared variables called *shunts*. Shunts are time-stamped with the time of the most recent write (the programmer does not have to worry about writing the time-stamp as it is assumed that the run-time execution environment will perform this task). Shunts may only be written to by a single agent throughout the lifetime of the system (although they may be read by many). Shunts are assumed to be non-blocking on reading and writing, and therefore an agent does not have to wait for a partner in order to read or write a shunt. Agents also have local protected state. The values found in shunts and variables

at the start of the system execution are nondeterministic. All agents are assumed to be terminating.

The use of timestamps in the shunts enables the user to reason about the freshness of data, and this, we believe, is one of the most important issues in real-time software design. Timestamps also provide the basic building block of real-time requirements specifications: we discuss this in detail in the next section.

3 The TAM Real-Time Logic

3.1 Overview

The TAM real-time logic is used both as a language in which to express requirements specifications, and as a formalism in which to define the semantics of the wide-spectrum language used in the TAM theory. It is constructed from conservative extensions to first-order predicate logic, and this enables the developer to use the standard first-order proof system. The logic formalises the notion of a *timed variable* which is the notation used to represent real-time program variables and shunts. Time is represented by positive integers, and a timing function is used to represent the values found in variables and shunts at a specific time. Specifications are therefore constraints on the relationship between time-stamps and values found in shunts during the lifetime of the system. Additional free variables are also provided which represent the release and termination time of the system; these variables may be predicated over in the usual way and therefore provide a mechanism for specifying duration.

The timing function is denoted '@', and is defined over pairs containing the name of a variable or shunt, and a time: thus the term '$@(X, 3)$' represents the value found in the variable (shunt) X at time=3. We usually write the term with '@' as an infix function. The projection function '$.ts$' and '$.v$' are also used to refer to the time-stamp and and value found in a shunt respectively, so we can write $s.v@t$ (the value found in shunt s at time=t), and $s.ts@t$ (the timestamp found in shunt s at time=t). The two free variables t_α and t_ω are used to denote the release time of the system, and termination time of the system respectively.

Example

Consider a simple real-time system which within 10 time units reads the integer value from a shunt called *in*, calculates the square of the number, and outputs the value to a shunt called *out*. It is assumed that the behaviour of the shunt *in* is constrained by the environment, but that the shunt *out* is entirely under the control of the system we are specifying. The liveness and timeliness property for this system is captured in the following formula:

$$\exists \sigma : \mathcal{N}(\sigma \in [t_\alpha, t_\omega] \wedge out.v@t_\omega = (in.v@\sigma)^2 \ \wedge \ t_\omega \leq t_\alpha + 10)$$

Of course it is also important that no other value is written to the shunt *out* during the execution of the system, and so we provide a safety requirement that asserts that during the execution of the system, there is only one write to the shunt *out* (we do this by counting the number of time-stamps which appear in *out* that are different to the time-stamp found at time=t_ω):

$$\#\{n | \exists \sigma : \mathcal{N}(\sigma \in [t_\alpha, t_\omega] \wedge out.ts@\sigma = n) \wedge n \neq out.ts@t_\omega\} \ = \ 1$$

The specification formed from the conjunction of these two formulae is much more complex than specifications commonly written for transformational systems. This is be-

cause we have to concern ourselves with the values found in the shunt *out* during the lifetime of the system rather than just at the start and end of execution; of course this is true of any specification language for reactive systems.

3.2 The TAM Logic Language

The real-time TAM logic is a multi-sorted, first-order, predicate logic, and is made up of the following symbols:

- The truth symbols *true* and *false*
- A set of variable symbols $x, y, z, ...$
- The duration symbols t_α and t_ω
- A set of computation variables $\mathbf{Name_v}$
- A set of computation variable variables $a, b, c, ...$
- A set of shunt names $\mathbf{Name_s}$
- A set of shunt name variables $s, s', ...$
- A set of constant symbols $\mathbf{C}$
- A set of function symbols $+, -, \times, ..$
- The timing function symbol '@' and the projection function symbols '$.ts@$' and '$.v@$'
- A set of predicate symbols $<, >, =, P, Q, ...$
- The propositional connectives $\wedge$ and $\neg$
- The universal quantifier $\forall$, and existential quantifier $\exists$
- Right and left parenthesis (,)

There are three sorts of terms, $\mathcal{N}$-terms, $Name_v$-terms and $Name_s$-terms. These terms are constructed as follows:

- The constant symbols in $\mathbf{C}$ are terms of sort $\mathcal{N}$
- The variable symbols are terms of sort $\mathcal{N}$
- The computation variable variables form terms of sort $Name_v$
- The shunt variables form terms of the sort $Name_s$
- The symbols in $\mathbf{Name_v}$ are terms of sort $Name_v$
- The symbols in $\mathbf{Name_s}$ are terms of sort $Name_s$
- If f is a function of arity=n and $t_1, .., t_n$ are terms of sort $\mathcal{N}$ then $f(t_1, .., t_n)$ is a term of sort $\mathcal{N}$
- If x is a term of sort $Name_v$ and t is a term of sort $\mathcal{N}$, then $@(x, t)$ is a term of sort $\mathcal{N}$

- If x is a term of sort $Name_s$ and t is a term of sort $\mathcal{N}$, then $.ts@(x,t)$ is a term of sort $\mathcal{N}$
- If x is a term of sort $Name_s$ and t is a term of sort $\mathcal{N}$, then $.v@(x,t)$ is a term of sort $\mathcal{N}$
- The duration symbols t_α and t_ω are terms of sort $\mathcal{N}$

Formulae are defined as follows:

- If P is an arity=n predicate, and $t_1, .., t_n$ are terms of the appropriate sort, then $P(t_1, .., t_n)$ is a formula
- If Φ and Ψ are formulae, then $\Phi \wedge \Psi$ is a formula
- If Φ is a formula then $\neg\Phi$ is a formula
- If x is a variable symbol which occurs free and of sort $Name_s$ in Φ, then $\forall x : Shunt(\Phi)$ is a formula (similarly for $exists$)
- If x is a variable symbol which occurs free and of sort $Name_v$ in Φ, then $\forall x : Var(\Phi)$ is a formula (similarly for $\exists$)
- If x is a variable symbol which occurs free and of sort $\mathcal{N}$ in Φ, then $\forall x : \mathcal{N}(\Phi)$ is a formula (similarly for $\exists$)

3.3 The Meaning of Real-Time TAM Logic Formulae

We define three sorts in our domain of interest: the set of positive integers $\mathcal{N}$, a set of text strings $\mathbf{Strings_s}$ and a set of strings $\mathbf{Strings_v}$. Functions and predicates are given their usual interpretation, and the terms are interpreted as follows:

- Each element in the set $\mathbf{C}$ is assigned an element in $\mathcal{N}$
- Each element in the set $\mathbf{Name_s}$ is assigned a unique text string in $\mathbf{Strings_s}$ which corresponds the the symbol found in name (e.g. $s \in Name_s$ is assigned the string 's')
- Each element in the set $\mathbf{Name_v}$ is assigned a unique text string in $\mathbf{Strings_v}$
- The duration symbols are assigned values from $\mathcal{N}$.
- The function '@' is assigned a value from the function space $[[\mathbf{Strings_s} \cup \mathbf{Strings_v} \times \mathcal{N}] \rightarrow \mathcal{N}]$
- The functions '$.ts@$' and '$.v@$' are each assigned a value from the function space $[[\mathbf{Strings_s} \times \mathcal{N}] \rightarrow \mathcal{N}]$

Given an interpretation $\mathcal{I}$ with the structure defined above, then we define satisfaction ($\models$) as follows:

$\mathcal{I} \models \Phi \wedge \Psi$	iff $\mathcal{I} \models \Phi$ and $\mathcal{I} \models \Psi$
$\mathcal{I} \models \neg\Phi$	iff not $\mathcal{I} \models \Phi$
$\mathcal{I} \models P(t_1, .., t_n)$	iff $\mathcal{I}[P](\mathcal{I}[t_1], .., \mathcal{I}[t_n])$
$\mathcal{I} \models \forall x : Shunt(\Phi)$	iff for every element s in $\mathbf{Strings_s}$ we have $\mathcal{I} \models \Phi[s/x]$
$\mathcal{I} \models \forall x : Var(\Phi)$	iff for every element v in $\mathbf{Strings_v}$ we have $\mathcal{I} \models \Phi[v/x]$
$\mathcal{I} \models \forall x : \mathcal{N}(\Phi)$	iff for every element n in $\mathcal{N}$ we have $\mathcal{I} \models \Phi[n/x]$
$\mathcal{I} \models \exists x : \mathcal{T}(\Phi)$	iff $\mathcal{I} \models \neg\forall x : \mathcal{T}(\neg\Phi)$ where $\mathcal{T}$ is $\mathcal{N}$, $Shunt$, or Var

We also assume the usual shorthand notation for defining disjunction, implication, and existential quantification. We use the '@' in infix form, and write $.ts@(s,t)$ as $s.ts@t$ and similarly for '$.v@$'. In addition we shall use the notation $s@n = s@m$ as a shorthand notation for $s.v@n = s.v@m \ \wedge \ s.ts@n = s.ts@m$, and the notation $s@n = (x, y)$ for $s.ts@n = x \ \wedge \ s.v@n = y$. We also assume the usual notation for indexed (finite) conjunction and disjunction.

3.4 Axioms

We can rely on the fact that writing to a shunt will cause the timestamp in the shunt to update appropriately.

(Freshness) $\forall s : Shunt(\forall t : \mathcal{N}(s.v@t \neq s.v@t - 1 \ \Rightarrow \ s.ts@t = t))$

We can also rely on the termination time of a system to be at, or after, the release time.

(Duration) $t_\omega \geq t_\alpha$

4 The TAM Language

The TAM language contains both a specification statement syntax, and a syntax for an imperative style real-time programming language, which we shall refer to as the *concrete* syntax. The major difference between the specification statement and the concrete syntax is that variables and shunts are referred to in the concrete syntax without reference to the timing function – the semantics are responsible for the association between the variables and shunts at the concrete level and the timed variables and shunts in the underlying logic. The syntax can be defined in terms of agents by the following table:

Agent form	Name
$w : \Phi$	Specification
$x := e$	Assignment
$(x, y) \leftarrow s$	Input
$x \rightarrow s$	Output
$\mathcal{A}; \mathcal{B}$	Sequence
$\bigsqcup_{i \in I} g_i \Rightarrow \mathcal{A}_i$	Conditional
$\mathcal{A} \mid \mathcal{B}$	Concurrent
$[S]\mathcal{A}$	Deadline
$\mathcal{A}/s$	Restriction
$(x)\mathcal{A}$	Local
$\mathcal{A} \triangleright_s^n \mathcal{B}$	Signal
$\mu_n \mathcal{A}$	Iteration

Where w is a set of shunt and variable names, Φ is a real-time TAM logic formula, e is some term on computation variables which evaluates to a value in $\mathcal{N}$, x and y are terms of sort $Name_v$, s is a term of sort $Name_s$, $\mathcal{A}$ and $\mathcal{B}$ are agents, I is some finite indexing set, g_i are boolean expressions (predicates) on shunt names and computational variable names, S is a set of values from $\mathcal{N}$, and n is a value from $\mathcal{N}$

The semantics and well-formedness conditions can be informally described as follows:

Specification

The user of the TAM theory is expected to keep account of the environment of each agent, i.e. the set of variable and shunt names which may be written to by each agent. The frame 'w' in the specification agent denotes those variables and shunts which are in the environment of the specification and which may have their value changed by an agent which is a valid refinement of the specification. Thus, any variable or shunt which is in the agent's environment, but which is not in the frame, can be assumed to remain stable during the execution of the agent.

The real-time TAM logic formula Φ provides a specification of the behaviour of the system which will eventually result from the refinement of $w : \Phi$. This is usually achieved by constraining the values and timestamps found in the shunts that appear in w, and by predicating over the values of t_α and t_ω.

Assignment

The expression e is a term within which variables may be referred to without reference to the timing function. This is because the application of the timing function with the time equal to the release time of the assignment agent will be assumed. Thus the agent form:

$$x := x + y$$

(where x and y are computation variable names) will be translated by the semantics into the constraint:

$$x@t_\omega = x@t_\alpha \ + \ y@t_\alpha$$

Note that because the variables x and y must belong to the assignment agent, and no concurrently executing agent can write to those variables it does not matter when the values are read, thus reading them at time=t_α is simply a notational convenience. The

assignment statement will also take some time in which to execute, and *no* assignment will be instantaneous (even assignments of the form $x := x$).

Input

The input statement reads the timestamp and value from a shunt at the same time. The timestamp is read into the left variable, and the value into the right. The read is asynchronous i.e. it does not need a partner (doing the writing) with which to synchronise. The reading cannot be instantaneous. The read occurs sometime between the release and termination of the read agent, but the user can not depend upon a particular instant (unless he constrains the reading further by deadlines and delays etc). If there has been no write to the shunt before the read takes place then the value and timestamp can have any value from $\mathcal{N}$.

Output

The output statement writes the value given into the shunt. The value can be any positive integer constant, or the value found in a computational variable. The write occurs sometime between the release time and termination time of the write agent, but the user can not depend upon a specific time (unless he specifically constrains the write with deadlines and delays etc). The run-time system is responsible for writing the current time as a timestamp into the shunt at the same time as the value is written. The writing is asynchronous in that the output agent does not have to wait for a partner (doing the reading) in order to write to the shunt. The shunt is assumed to have been written to before the output agent terminates.

Sequence

The termination time of the first agent becomes the release time of the second. Thus sequencing defines a precedence relation not a physical operation, and the transfer of control is assumed to occur instantaneously. The final instant of the first agent is also the first instant of the second, and this initially suggests the scheduling problem of the 'dangling drop' (i.e. the extreme case where the first task is pre-empted at the point where it has actually completed all of its computation, but has not yet told the run-time system; technically the deadline of the first task may be missed even though it has completed all of its useful work). This however is not a problem as an idle delay tick may be inserted at the release time of the second agent. This is consistent with the semantics: no agent may do any communication on its' release instant and so it can be assumed to idle.

Conditional

Each of the boolean expressions are evaluated, and the agent corresponding to a true value will be executed immediately. The expressions can be any predicate (with shunts and variable names occurring untimed). If none of the boolean expressions are true then the agent terminates immediately, and if more than one is true then a non-deterministic choice is made between them. The semantics do not assume that the evaluation of the conditionals requires any computational resources, and this enables the user to encode idle polling. However, care needs to be taken during refinement to ensure that computationally expensive evaluation is not constrained into an infeasibly small interval; this is discussed in more detail in section 6. As with the assignment, the values at the release time of the agent are used (i.e. shunt values at the release time of the conditional agent are used – this must also be taken into account when conditional evaluation is considered). We use the notation $g \Rightarrow \mathcal{A}$ when only one conditional is present.

Concurrency

Concurrency is distributed and so the concurrent composition terminates when both

agents have terminated. Concurrent agents should not write to the same shunt, and should not refer to the same computation variables. An attempt to do so may result in the system aborting with unpredictable results. We use the shorthand notation $\prod_{i \in I} \mathcal{A}_i$ for indexed concurrency. Concurrency should be seen as a declaration of the lack of precedence constraints between two agents: if those agents are representing tasks then the concurrency operator declares that those two tasks can be independently scheduled.

Deadline

It is assumed that the duration of the agent (the difference between release and termination time) is equal to one of the values found in the set S. Thus the deadline agent forms a constraint upon the run-time execution environment that the refinement calculus and the system developer can depend upon being met.

Restriction

The shunt in the restriction becomes 'hidden' from the rest of the system and may only be written and read by the agent specified. This operator becomes useful when program equivalence is being proven: it is possible to prove that a program which uses a number of concurrently executing agents, communicating via hidden shunts, is equivalent to an agent with no apparent internal structure.

Local

The variable in the local declaration becomes 'hidden' and the rest of the system (i.e. the sequential agents which follow the agent with the hidden variable) may not read it or write to it. We use the shorthand notation $(x, y)\mathcal{A}$ in place of $(x)(y)\mathcal{A}$ etc. for clarity.

Signal

The given shunt is treated as a signal, and is monitored from the release time for the number of time units specified. If the shunt is written to in that interval then the agent on the right is released with a release time equal to that of the first write to the shunt, otherwise the agent on the left is released at the end of the interval.

Iteration

The specified agent will be executed in sequence the given number of times. This agent is simply used as a shorthand for long sequences of task executions.

4.1 Semantics

We start by defining some useful predicates on shunts and variables. The predicate '*stable*' asserts that the shunt s will not be changed during the given interval:

Definition Stable (shunts) $stable(s, n, m) =_{def} \bigwedge_{\sigma \in (n,m]} s@\sigma = s@(\sigma - 1)$

Similarly for variables:

Definition Stable (variables) $stable(x, n, m) =_{def} x@m = x@n$

In addition, the definitions for *stable* are extended to sets of variables or shunts.

The predicate *write* asserts that a given value is written to a shunt within an interval, and that the shunt remains stable at all other times within the interval:

Definition Write $write(x, s, n, m) =_{def}$

$$\bigvee_{\sigma \in (n,m]} stable(s, n, \sigma - 1) \ \wedge \ s@\sigma = (\sigma, x@n) \ \wedge \ stable(s, \sigma, m)$$

$$[\![w:\Phi]\!] =_{def} stable(\tilde{w}, t_\alpha, t_\omega) \wedge \Phi \qquad [\![x := e]\!] =_{def} [\![\{x\} : (t_\alpha < t_\omega \wedge x@t_\omega = e@t_\alpha)]\!]$$

$$[\![x \rightarrow s]\!] =_{def} [\![\ \{s\} : write(x, s, t_\alpha, t_\omega)]\!] \qquad [\![\mathcal{A}/s]\!] =_{def} \exists x : Shunt([\![\mathcal{A}]\!][x/s])$$

$$[\![(x, y) \leftarrow s]\!] =_{def} [\![\{x, y\} : t_\alpha < t_\omega \wedge \exists m : \mathcal{N}(m \in (t_\alpha, t_\omega] \wedge x@t_\omega = s.ts@m \wedge y@t_\omega = s.v@m)]\!]$$

$$[\![(x)\mathcal{A}]\!] =_{def} \exists y : Var([\![\mathcal{A}]\!][y/x])$$

$$[\![\mathcal{A}|\mathcal{B}]\!] =_{def} ([\![\mathcal{A}]\!]^\frown[\![\mathbb{1}]\!]) \wedge ([\![\mathcal{B}]\!]^\frown[\![\mathbb{1}]\!])$$

$$[\![[S]\mathcal{A}]\!] =_{def} [\![\mathcal{A}]\!] \wedge t_\omega - t_\alpha \in S \qquad [\![\mathcal{A}; \mathcal{B}]\!] =_{def} [\![\mathcal{A}]\!]^\frown[\![\mathcal{B}]\!]$$

$$[\![\bigsqcup_{i \in I} g_i \Rightarrow \mathcal{A}_i]\!] =_{def} (((\bigwedge_{i \in I} \neg g_i@t_\alpha) \wedge [\![\mathbb{1}]\!]) \vee \bigvee_{i \in I} (g_i@t_\alpha \wedge [\![\mathcal{A}_i]\!])))$$

$$[\![\mathcal{A} \triangleright^s_n \mathcal{B}]\!] =_{def} \text{Nowrite}^\frown[\![\mathcal{A}]\!] \ \vee \ \text{Input}^\frown[\![B]\!]$$

$$[\![\mu_{n+1}\mathcal{A}]\!] =_{def} ([\![\mathcal{A}]\!])^\frown[\![\mu_n\mathcal{A}]\!] \qquad [\![\mu_0\mathcal{A}]\!] =_{def} [\![\mathbb{1}]\!]$$

$$\text{Nowrite} =_{def} [\![\emptyset : t_\omega = t_\alpha + d \ \wedge \ s.ts@t_\alpha < t_\alpha \ \wedge \ stable(s, t_\alpha, t_\omega)]\!]$$

$$\text{Input} =_{def} [\![\emptyset : t_\omega \in [t_\alpha, t_\alpha + d] \ \wedge \ s.ts@t_\omega = t_\omega \ \wedge \ \forall m \in [t_\alpha, t_\omega)(s@m = s@t_\alpha)]\!]$$

Figure 1: TAM Semantics

We also define an operator for dividing formulae into time consecutive subformulae:

Definition Chop. Given two timed logic formulae $\mathcal{A}$ and $\mathcal{B}$, then,

$$\mathcal{A}^\frown\mathcal{B} =_{def} \exists m : \mathcal{N}(m \in [t_\alpha, t_\omega] \wedge \mathcal{A}[m/t_\omega] \wedge \mathcal{B}[m/t_\alpha])$$

The semantics of an agent are now given by a timed logic formula. The specification statement is defined in this manner also, giving a natural interpretation for a refinement relation. Note that we use the notation $\tilde{w}$ to denote those variables and shunts which are owned by the specification agent, but which do not appear in the frame, and the notation $\mathbb{1}$ to denote the agent $\emptyset : true$ (a particularly useful agent). Figure 1 gives the definition of the semantics. We also assume that the set of variables and shunts owned by the agent $\mathcal{A}|\mathcal{B}$ is partitioned into disjoint environments for $\mathcal{A}$ and $\mathcal{B}$; the identification of environments with agents is informal, and it is the responsibility of the refiner to decide upon suitable partitions[3].

5 Standard Agent Definitions

In a number of TAM publications, definitions for agents which capture real-time behaviour succinctly have been proposed, these definitions have been syntactic sugar for complex concrete agent forms. In this section we define the standard agent forms for periods, delays, finding the current time (within bounds), and managing mutual exclusion.

[3]The partioning is almost always obvious as the frame of any specification agent dictates the minimum contents of the specification agent's environment. Those variables or shunts which are not changed by either of the two concurrently executing agents may appear in either environment.

5.1 Minimum Delays

A delay agent does not change any variable or shunt value, and guarantees not to terminate for a minimum duration (i.e. models an idle delay). Given a duration $n \in \mathcal{N}$, then we define the delay agent as:

$$\delta n \ =_{def} \ \emptyset : t_\omega \geq t_\alpha + n$$

A specific length delay (which might be used to model task offsets for instance) can be defined by:

$$\Delta n \ =_{def} \ [\{n\}]\delta n$$

This agent will guarantee to terminate at exactly n time units after release.

5.2 The Current Time

Consider the following agent which writes to a private shunt and then reads the timestamp which was just written:

$$(0 \rightarrow s; (ts, x) \leftarrow s)/s; \mathcal{A}$$

The value found in the variable ts will provide a *lower bound* only on the current time. This is because the agent may have been pre-empted between the writing of the shunt and the reading, or between the reading and the use of the timestamp in agent $\mathcal{A}$.

However, consider the following agent which performs the same task, but within a tight deadline:

$$[m]((0 \rightarrow s; (ts, x) \leftarrow s; \mathcal{A})/s)$$

The user knows that in the agent $\mathcal{A}$, the value found in the variable ts will have a bound on freshness, i.e. he will know that the current time is somewhere between ts and $ts + m$.

5.3 Specific Deadlines

We overload the deadline operator to define a more restricted deadline. We use the notation $[n]\mathcal{A}$ (where n is a single positive integer value) to denote the agent $[\ [1..n]\ ]\mathcal{A}$.

5.4 Periodic Agents

The periodic agent can now be defined. Given an agent $\mathcal{A}$, a period=T, a deadline=D, and a number of periods=n, then:

$$Period(\mathcal{A}, T, D, n) \ = \ \mu_n([T]([D]\mathcal{A}|\delta T))$$

In this definition we assume that $D \leq T$, if we remove this constraint then we have a more general periodic agent:

$$Period(\mathcal{A}, T, D, n) \ = \ \prod_{i \in [1..n]} \Delta(T \times i); [D]\mathcal{A}$$

But note that in overlapping agents we have to consider certain constraints on the possibility in the overlap of writing to the same shunt, this is discussed in more detail in section 7.

5.5 Critical Sections

Consider the case when two (or more) agents wish to write to the same shunt (i.e. share the same resource), and the system has to enforce mutual exclusion. We can model this by attaching a *semaphore-dispatch* agent to the shared shunt which communicates to the requesting tasks via three sets of shunts: Req_1, Req_2 are written to by the requesting agents when they require exclusive access to the shunt, Gnt_1, Gnt_2 are used to grant access (the semaphore-dispatch agent writes to the shunt of the agent requesting access), and Rel_1, Rel_2 are used to release the semaphore and are written by the requesting agent when it has finished. We also assume a priority of agent 1 over agent 2 if two requests for access are made at the same time. The semaphore-dispatch agent is assumed to grant up to n semaphores (or ticks for which the semaphore is not granted) before it removes the resource from the system. We also place a bound on the waiting time for the release signal of m time units (in case an agent 'hangs' without releasing): this is acceptable only if we choose an m which is greater than the deadline of both requesting agents.

We can write the semaphore-dispatch agent as follows:

$$\mu n((\Delta 1 \triangleright_0^{Req_2} [1](0 \rightarrow Gnt_2); R(2)) \triangleright_0^{Req_1} [1](0 \rightarrow Gnt_1); R(1))$$

$$R(n) \ =_{def} \ \Delta 0 \triangleright_m^{Rel_n} \Delta 0$$

The behaviour of this agent can be read as follows. The first signal to be tested is that for Req_1 (the request for access from agent 1), the test will only be for a single instant in time. If the agent is signalling then the grant shunt will be written to within one tick, and the semaphore-dispatch agent then behaves like $R(1)$ which is waiting for the release from agent 1. When the release comes (or the signal times out) the semaphore-dispatch agent returns to the waiting state. If the signal from agent 1 is not written then the signal from agent 2 is tested, if it is being written then the same behaviour occurs as for agent 1, if it is not being written then the sempahore-dispatcher waits for the next tick and starts the signal monitoring again. Note that semaphore requests which occur during the wait for a release signal will be ignored, and thus signalling must be repeated by the requesting agent periodically. Also, the grant signal time is bounded by the deadline on the grant write (in this instance this is a single tick).

Note that this semaphore-dispatcher method of dealing with mutual exclusion is only one of a possible number of solutions to the problem. Consider the solution whereby requests are not lost, and are guaranteed to be serviced when the resource next becomes free. This is achieved by the semaphore-dispatcher updating local variables rt_1 and rt_2 with the timestamp of the most recent request of each agent:

$$\begin{array}{l}(x, y, rt_1, rt_2)(\ [1]((rt_1, x) \leftarrow Req_1 | (rt_2, y) \leftarrow Req_2); \\ \qquad\qquad \mu_n \ (\ Req_1.ts > rt_1 \Rightarrow P(1) \\ \qquad\qquad\qquad \sqcup \\ \qquad\qquad\qquad Req_2.ts > rt_2 \Rightarrow P(2) \\ \qquad\qquad\qquad \sqcup \\ \qquad\qquad\qquad Req_1.ts = rt_1 \wedge Req_2.ts = rt_2 \Rightarrow \Delta 1 \\ \qquad\qquad)) \end{array}$$

$$P(1) \ =_{def} \ [1]((rt_1, x) \leftarrow Req_1); [1](0 \rightarrow Gnt_1); (\Delta 0 \triangleright_m^{Rel_1} \Delta 0)$$

$$P(2) \ =_{def} \ [1]((rt_2, y) \leftarrow Req_2); [1](0 \rightarrow Gnt_2); (\Delta 0 \triangleright_m^{Rel_2} \Delta 0)$$

In this solution there is idle waiting on the release signals, and on the request signals

(in this special instance – where the conditions are expressions on shunt timestamps – we can assume that a conditional agent which does nothing but execute an idle delay is implemented by an agent which idles until the condition holds). Note that again, the deadlines on grant signals could be slackened if necessary.

6 The Refinement Calculus

The aim of a refinement calculus is to provide a set of syntactic rewrite rules which enables the software developer to transform a requirements specification into an executable program. The calculus must ensure that any program resulting from the application of laws must be correct with respect to the original specification. Correctness is defined in terms of a refinement relation, and individual refinement laws are proven sound with respect to this relation.

We define a refinement relation as follows:

Definition Refinement. Given two agents $\mathcal{A}$ and $\mathcal{B}$, then $\mathcal{B}$ refines $\mathcal{A}$ (written $\mathcal{A} \sqsubseteq \mathcal{B}$) exactly when $[\![\mathcal{B}]\!] \Rightarrow [\![\mathcal{A}]\!]$.

Refinement can be seen as a lessening of nondeterminism, i.e. given a specification which lists a number of acceptable alternatives (which might be inherent in the underspecification of a system), then a program which guarantees at least one of those alternatives is a valid refinement.

It is clear that the refinement relation is a partial order (a property inherited from the implication connective), and this will allow us to perform refinement steps of any granularity without affecting the resulting program.

If we return to our early example of a system specification, we are now in a position to prove the refinement:

$$\begin{aligned}\{out\} :& \exists \sigma : \mathcal{N}(\sigma \in [t_\alpha, t_\omega] \wedge out.v@t_\omega = (in.v@\sigma)^2 \wedge t_\omega \leq t_\alpha + 10) \\ & \wedge \#\{n | \exists \sigma \in \mathcal{N}(\sigma \in [t_\alpha, t_\omega] \wedge out.ts@\sigma = n) \wedge n \neq out.ts@t_\omega\} \;=\; 1 \\ \\ & \sqsubseteq \quad (x, y)[10]((x, y) \leftarrow in; y^2 \rightarrow out)\end{aligned}$$

The proof sketch is as follows:

$$\begin{aligned}&[\![(x, y)[10]((x, y) \leftarrow in; y^2 \rightarrow out)]\!] \\ &\qquad = \exists x, y : Var(\\ &\qquad\quad \exists m : \mathcal{N}(m \in [t_\alpha, t_\omega] \wedge \bigvee_{\sigma \in (t_\alpha, m]} (x@t_\omega = in.ts@\sigma \wedge y@t_\omega = in.v@\sigma) \\ &\qquad\qquad\qquad\qquad \wedge stable(out, t_\alpha, m) \\ &\qquad\qquad\qquad\qquad \wedge write(y@m^2, out, m, t_\omega) \wedge t_\omega \leq t_\alpha + 10)\end{aligned}$$

We can see that:

$$\begin{aligned}&stable(out, t, m) \wedge write(y@m^2, out, m, t) \Rightarrow \\ &\quad \#\{n | \exists \sigma : \mathcal{N}(\sigma \in [t_\alpha, t_\omega] \wedge out.ts@\sigma = n) \wedge n \neq out.ts@t_\omega\} \;=\; 1\end{aligned}$$

by the definition of *stable* and *write*. The liveness property is guaranteed by the theorem:

$$\exists x, y : Var(\exists m : \mathcal{N}(m \in [t_\alpha, t_\omega] \wedge \bigvee_{\sigma \in (t_\alpha, m]} (x@t_\omega = in.ts@\sigma \wedge y@t_\omega = in.v@\sigma) \wedge write(y@m^2, out, m, t_\omega) \wedge t_\omega \leq t_\alpha + 10)$$

$$\Rightarrow \exists \sigma : \mathcal{N}(\sigma \in [t_\alpha, t_\omega] \wedge out.v@t_\omega = (in.v@\sigma)^2 \wedge t_\omega \leq t_\alpha + 10) \qquad \Box$$

It would be infeasible to prove refinements of any reasonable sized program in this manner, the complexity of such a task would ensure that mistakes would be made. Instead we provide a calculus of refinement laws which have much simpler proof obligations. We define a subset of these laws below, each law is labelled so that it can be referred to in refinement proofs.

We start with a refinement law schema which arises from the semantic definitions.

RS (refinement schema) If $[\![\mathcal{A}]\!] =_{def} \Phi$ then $\Phi \sqsubseteq \mathcal{A}$

We now define the laws of the calculus by construct, we use the equality symbol '=' to denote that refinement is valid in both directions.

Specification

SR.1 $w : \Phi \;=\; w : \Phi[x@t_\alpha / x@t_\omega]$ (if $x \notin w$)

SR.2 $w : \Phi \;=\; w : \Phi[x@t_\omega / x@t_\alpha]$ (if $x \notin w$)

SR.3 $w \cup \{s\} : \Phi \;\wedge\; stable(s, t_\alpha, t_\omega) \;=\; w - s : \Phi$

(if s not in Φ)

SR.4 $w : \Phi \;\sqsubseteq\; w : \Phi'$ (if $\Phi' \equiv\rangle \Phi$)

Restriction

For any variable x:

$VR.1\ (x)\mathcal{A} \;=\; \mathcal{A}$ (if x not in $\mathcal{A}$)

$VR.2\ (x)((y)\mathcal{A}) \;=\; (y)((x)\mathcal{A})$

$VR.3\ w : (\exists y : Var(\Phi)) \;=\; (x)(w \cup \{x\} : \Phi[x/y])$ (if $x \notin w : \Phi$)

$VR.4\ w : \Phi \;\sqsubseteq\; (x)w \cup \{x\} : \Phi$

(if x is new unique computational variable)

Sequential

SE.1 $P;(Q;R) \;=\; (P;Q);R$

Deadline

DE.1 $[S]\mathcal{A} \sqsubseteq [S']\mathcal{A}$ (if $S' \subseteq S$)

Delay

DL.1 $\delta n; \delta m \;=\; \delta n + m$ DL.2 $\Delta n; \Delta m \;=\; \Delta n + m$

DL.3 $\Delta n; \perp\!\!\!\perp = \delta n$ DL.4 $\delta 0 = \perp\!\!\!\perp$

Concurrent

CR.1 $\mathcal{A}|\mathcal{B} = \mathcal{B}|\mathcal{A}$ CR.2 $\mathcal{A}|(\mathcal{B}|\mathcal{C}) = (\mathcal{A}|\mathcal{B})|\mathcal{C}$

Conditional

GR.1 $\bigsqcup_{i\in I} g_i \Rightarrow \mathcal{A}_i \sqsubseteq \mathcal{A}_j$ (if $g_j \equiv true$)

Signal

For any shunt s and time n:

TR.1 $(\mathcal{A} \triangleright_n^s \mathcal{B})|(\mathcal{C} \triangleright_n^s \mathcal{D}) = (\mathcal{A}|\mathcal{C}) \triangleright_n^s (\mathcal{B}|\mathcal{D})$

TR.2 $\mathcal{A} \triangleright_{n+1}^s \mathcal{B} = (\mathcal{A} \triangleright_n^s \mathcal{B}) \triangleright_1^s \mathcal{B}$

TR.3 $\mathcal{A} \triangleright_n^s (\mathcal{C} \triangleright_0^s \mathcal{B}) = \mathcal{A} \triangleright_n^s \mathcal{B}$

Iteration

IR.1 $\mu_{n+1}\mathcal{A} = \mathcal{A}; \mu_n\mathcal{A}; \perp\!\!\!\perp = \mu_n\mathcal{A}; \mathcal{A}; \perp\!\!\!\perp$

Of course there are many such useful rules, and the reader can probably think of many more. The refinement calculus needs to be proven sound with respect to the definition of the refinement relation, and the proof is presented in various of the later TAM papers. In addition, the refinement relation needs to be proven monotonic, i.e.

Monotonicity Given that $\mathcal{A} \sqsubseteq \mathcal{B}$, then for any context $\mathcal{C}[_]$, we have $\mathcal{C}[\mathcal{A}] \sqsubseteq \mathcal{C}[\mathcal{B}]$.

This property enables the user of the calculus to remove an agent from its context, refine it isolation, and replace the refined agent back into the same context without requiring them to proving that the new composed agent remains a valid refinement. The proof of this theorem is given in [6].

7 Postscript: A Few Notes on 'Sensible' Refinements

It is possible to refine any agent by the specification $w : false$ (for any frame w), this is because:

$$\begin{aligned}[\![w : false]\!] &= stable(\tilde{w}, t_\alpha, t_\omega) \wedge false \\ &= false \\ &\Rightarrow [\![\mathcal{A}]\!] \text{ (any } \mathcal{A}) \ \Box\end{aligned}$$

The specification $w : false$ is often referred to as the miraculous specification (see [2] for example), and usually arises as a result of an inconsistent specification at an earlier stage in refinement. However,other kinds of refinement can result in undesired agents as well. Consider the following refinement:

$$\{out\} : write(1, out, t_\alpha, t_\omega)$$

$$\sqsubseteq \{out\} : write(1, out, t_\alpha, t_\omega) \wedge write(1, out, t_\alpha, t_\omega) \qquad \text{(by 'strengthen')}$$
$$\sqsubseteq \{out\} : \exists m : \mathcal{N}(m \in [t_\alpha, t_\omega] \wedge write(1, out, t_\alpha, m) \wedge stable(out, m, t_\omega))$$
$$\wedge \exists l : \mathcal{N}(l \in [t_\alpha, t_\omega] \wedge write(1, out, t_\alpha, l) \wedge stable(out, l, t_\omega))$$
(by 'strengthen')
$$\sqsubseteq \{out\} : write(1, out, t_\alpha, t_\omega) | \{out\} : write(1, out, t_\alpha, t_\omega) \text{ (by } \mathbf{RS})$$
$$\sqsubseteq 1 \rightarrow out | 1 \rightarrow out \text{ (by } \mathbf{RS})$$

In the final concurrent agent we expect that an implementation would be able to write the value to the shunt *out* twice, and *independently* – but the semantics of the agent dictate that the two agents must write at the same instant in time (otherwise the stability constraints inherent in the definition of the predicate *write* would be contradicted). It is important to realise that the refinement is not incorrect, but that in this case the informal understanding of the meaning of the concurrent agent is not supported by the actual semantics[4].

In order to make sure that our informal view is supported as much as is feasible by the refinement calculus we can introduce a number of heuristics. For example we can assert the following heuristic rule:

Separation Constraint The refinement $w : \Phi \sqsubseteq w_{\mathcal{A}} : \Phi_{\mathcal{A}} | w_{\mathcal{B}} : \Phi_{\mathcal{B}}$ is acceptable only when $w_{\mathcal{A}} \cup w_{\mathcal{B}} = w$ and when $w_{\mathcal{A}} \cap w_{\mathcal{B}} = \emptyset$

This heuristic would have enabled the developer to reject the above refinement in favour of a more 'sensible' alternative. Of course we could have defined the original refinement law for concurrency with this constraint enforced, but we believe that the underlying theory should be as flexible as possible, and that such high-level pragmatics should be defined at as late a stage as possible.

Another instance of a 'sensible' refinement hueristic is that of allowing time for the evaluation of conditionals. The semantics do not assume that the conditionals require evaluation time, and this enables the user to define programs which block on events. Consider the following agent:

$$\mu_n(s.v = 1 \Rightarrow \mathcal{A} \ \sqcup \ s.v \neq 1 \Rightarrow \Delta 1)$$

This agent is waiting for the value in the shunt s to be set to '1': if the value is not currently set to '1' then the agent idle waits for one tick and then tries again. When the value is set to 1 the agent $\mathcal{A}$ will be released. Note that if the owner of the shunt s does not change the value in s then the agent $\mathcal{A}$ might be released many times (although they will be precedence constrained).

We can see that one of the properties of this agent is that if the shunt s at the release time does not have the value 1, and the shunt remains stable, then the agent will reduce to an idle delay, i.e. we can assert the theorem:

$$\mu_n(s.v = 1 \Rightarrow \mathcal{A} \ \sqcup s.v \neq 1 \Rightarrow \Delta 1) \quad \sqsubseteq \quad \delta n \quad (\text{if } s.v@t_\alpha \neq 1 \wedge stable(s, t_\alpha, t_\alpha + n))$$

Proof (by induction)

(Base case where n=0)

(1) $\mu 0(s.v = 1 \Rightarrow \mathcal{A} \sqcup s.v \neq 1 \Rightarrow \Delta 1) \ = \ \mathbb{1}$ (by def Iteration)

[4]This is an excellent example of why it is important that any compiler for TAM – or any other formal development language – must also be proved correct: it is not good enough that we believe that the compiler supports our informal understanding of the semantics of the language.

(2) $= \delta 0$ (by DL.4)

(Inductive step)

If $\mu_n(s.v = 1 \Rightarrow \mathcal{A} \sqcup s.v \neq 1 \Rightarrow \Delta 1) \sqsubseteq \delta n$

then $\mu_n + 1(s.v = 1 \Rightarrow \mathcal{A} \sqcup s.v \neq 1 \Rightarrow \Delta 1) \sqsubseteq \delta n + 1$

(1) $\mu_n + 1(s.v = 1 \Rightarrow \mathcal{A} \sqcup s.v \neq 1 \Rightarrow \Delta 1)$
$= \mu_n(s.v = 1 \Rightarrow \mathcal{A} \sqcup s.v \neq 1 \Rightarrow \Delta 1); (s.v = 1 \Rightarrow \mathcal{A} \sqcup s.v \neq 1 \Rightarrow \Delta 1); \mathbb{1}$ (by IR.1)
(2) $\delta n + 1 = \delta n; \delta 1$ (by DL.1)
(3) $(s.v = 1 \Rightarrow \mathcal{A} \sqcup s.v \neq 1 \Rightarrow \Delta 1) \sqsubseteq \Delta 1$ (by def Conditional and stability constraint)
(4) $(s.v = 1 \Rightarrow \mathcal{A} \sqcup s.v \neq 1 \Rightarrow \Delta 1); \mathbb{1} \sqsubseteq \Delta 1; \mathbb{1}$ (by monotonicity)
(5) $\Delta 1; \mathbb{1} \sqsubseteq \delta 1$ (by 1,3 and DL.3)

and so by monotonicity the inductive step holds □

Thus the implementation would be sensible. However, it would be difficult to implement a conditional which relyed on computation, and also relyed on a very short deadline. Consider the agent:

$[m](x^y > z \Rightarrow (\delta m | \mathcal{A}))$

which gives a deadline of m time units, all of which is required for the execution of the agent $\mathcal{A}$: thus, the conditional must be evaluated instantaneously. This may be implementable by using some of the time given to the agent $\mathcal{A}$, but this certainly should not be relyed upon.

8 Conclusion

We have defined a standard for the TAM theory, including a syntax, semantics, and a refinement calculus. We have also given a number of examples of how the language might be used to express various real-time implementation issues. Finally, we have discussed how the refinement calculus might be used in a realistic manner in order to develop programs which are implementable.

Clearly there are many facets of the language yet to investigate, both in the existing constraints of the theory, and in the possibility of extending the theory to deal with issues such as non-termination, and type refinement, etc. The standard offered in this paper should provide a firm foundation for these experiments.

Acknowledgements

The authors would like to thank the Real-Time research group at York for their helpful comments, and in particular Ken Tindell for his reference to the 'dangling drop' problem of pre-emption, and discussions on the expressiveness of the computational model.

References

[1] Faron Moller and Christopher M N Tofts. A temporal calculus of communicating systems. Technical Report ECS-LFCS-89-104, University of Edinburgh, December 1989.

[2] C. Morgan. *Programming from Specifications*. Prentice-Hall, 1990.

[3] M. Portman and H. S. M. Zedan. The development of imprecise real-time systems. *Systems and Software*, (to appear 1993).

[4] A. Salwicki and T. Muldner. *On the Algorithmic Properties of Concurrent Programs*, volume LNCS 125. Springer-Verlag, 1981.

[5] S. Schneider. *Correctness and Communication in Real-Time Systems.* PhD thesis, 1990.

[6] D. J. Scholefield. *A Refinement Calculus for Real-Time Systems.* PhD thesis, The University of York, 1992.

[7] D. J. Scholefield and H. S. M. Zedan. The refinement of real-time systems. In *Fourth European Workshop on Real-Time Systems*, Athens, Greece, June 1992. IEEE Computer Society Press.

[8] D. J. Scholefield and H. S. M. Zedan. Tam: A formal framework for the development of distributed real-time systems. In *Symposium on Formal Techniques in Real-Time and Fault Tolerant Systems*, Nijmegen, Netherlands, January 1992.

[9] D. J. Scholefield and H. S. M. Zedan. Weakest precondition semantics for time and concurrency. *Information Processing Letters*, 43:301–308, October 1992.

[10] D. J. Scholefield, H. S. M. Zedan, and J. He. A predicative semantics for the refinement of real-time systems. In *Proceedings of MFPS IX, New Orleans 1993.* Springer-Verlag, (to appear) 1993.

[11] D. J. Scholefield, H. S. M. Zedan, and J. He. Real-time refinement: Semantics and application. In *Proceedings of MFCS 93, Gdansk 1993*, (to appear) 1993.

[12] D. J. Scholefield, H. S. M. Zedan, and J. He. A specification oriented semantics for the refinement of real-time systems. *Theoretical Computer Science*, 130, August 1994.