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, a refinement calculus is presented and a number of examples are discussed.

1 Introduction

1.1 Overview

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) [SZ91] [SZ92], 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 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.

1.2 Formal Methods for Specification and Design

A common criticism of formal theories for software development is the lack of method to accompany the often obscure notation and verification system. In many cases the users are not given advice as to which development process should be tackled first and how to verify work once it has been completed. Under such circumstances it is not surprising that little confidence is held in the final product.

If we examine case studies both in industrial projects and academic work involving formal theories, it soon becomes apparent that there often *is* an implicit development method discovered or formulated by the user. In this section we 'tease out' two major development methods and comment on their advantages and disadvantages. We also discuss how some example formal development theories use these methods.

Our major objective is to describe the method adopted for real-time systems development in the TAM theory, and how the TAM language supports the different stages within the method.

1.2.1 Development Stages

In order to discuss development methods, four different system descriptions are defined. The descriptions result from different processes within the methods. The definitions we give are not necessarily representative of the terms commonly used in the literature (which are either inconsistent or undefined).

We define the four description kinds as follows:

- Informal Requirements Specification. This is a client's view of the required system, usually written in plain English, or a mixture of English, mathematical modelling, and numerical data etc. We denote this description kind as $RS(i)$.
- Formal Requirements Specification. Which is a formal (or at least rigorous) description of the $RS(i)$ amenable to formal analysis and consistency checks. The description is usually in a form which abstracts away from resource considerations or partitioning, but which may contain non-functional requirements such as timing and criticality. We denote the formal requirements specification as $RS(f)$.

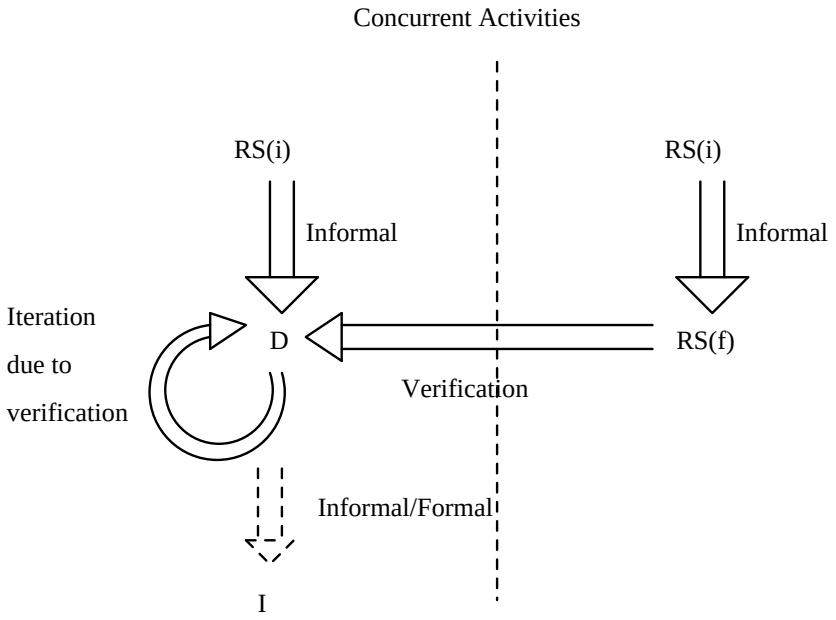

Figure 1: The Complement Method

- Design. Which is a description of the partitioning and algorithmic details of the $RS(f)$ which possibly includes information on flow-control (e.g. concurrency and sequential precedence constraints), and modularisation etc. We denote the design as D.
- Implementation. Which is a description of the system which may be executed, or at least understood and manipulated, by a computer. We denote the implementation as I.

Of course, the description kinds listed above are very simplistic and in many development methods the distinctions often become blurred or in extreme cases interchanged. Very rarely does a method result in these orthogonal description forms, but they do provide a starting point from which we may investigate the methods and theories.

1.2.2 The Complement Method

The most common development method for formal theories can be called the *complement* method. In this method it is usual to have two independent languages (with distinct syntax and semantics) with which to formulate the $RS(f)$ and D descriptions. A verification relationship is defined between the $RS(f)$ and D languages.

The complement method starts with two distinct procedures. In the first, the $RS(i)$ is translated into an $RS(f)$ in a necessarily informal manner. In the second, the $RS(i)$ is translated into a design (D) (also informally). The verification relation is then used to prove the design and $RS(f)$ are consistent, and if they are not, the design is iteratively amended until the verification relation holds. Then the design is translated (either informally or formally) into an implementation (I). Figure 1 shows how the 'idealised' complement method works.

Examples of theories which implicitly or explicitly use this method are given in the following table, along with their representation languages:

Theory	$RS(f)$	D	I	Verification	Ref
TCSP	Predicate Logic (traces)	CSP Proc Alg	OCCAM	sat	[Schn90]
TCCS	Process Logic	CCS Proc Alg	CWB	sat	[TM89]
RTTL	Temporal Logic	ESMs	Conic	|=	[Ost89]

There are a number of important advantages inherent in the complement method:

- Intuitive. Users often have a strong preference for starting software development with the design. This can provide insights into the $RS(i)$ and helps to highlight obvious problems. Supporting the users' intuition within a method in this manner probably reduces the number of errors in the informal translation stages ($RS(i) \Rightarrow D$ and $RS(i) \Rightarrow RS(f)$).
- Self-checking. By repeating the translation from the $RS(i)$ to the formal descriptions (both D and $RS(f)$) provides some fault tolerance within the method. This is particularly true if the two translations are undertaken by different parties.
- Efficient. The design description may be developed with efficiency in mind, the user has a free hand in design decisions.
- Re-use. Modules may be easily re-used, simplifying the design stage.

There are also a number of disadvantages:

- Iteration. The iteration caused by developing designs which are incorrect with respect to the $RS(f)$ may be time consuming. There is usually little guidance on how to amend the design in order to overcome specific verification problems.
- Complexity. The user needs to understand two different languages and the verification relation between them. This problem also has repercussions on development automation, which would require two logically separate software systems to cope with the two languages.
- Informality. There are two informal translation processes in the complement method. These are prone to error.

1.2.3 Refinement

The refinement method has gained much more credibility in recent years with a number of supporting theories being used in substantial case studies.

Refinement requires a *wide spectrum* language in which both $RS(f)$ and D descriptions may be formulated. The first translation, from $RS(i)$ to $RS(f)$, is necessarily informal. The $RS(f)$ description uses a small subset of the language's syntax (those structures which could be said to be more 'abstract'). A refinement relation defines those transformations used in both the $RS(f) \Rightarrow RS(f)/D$ process, and the $RS(f)/D \Rightarrow RS(f)/D$ process. Refinement transformations continue until a purely a concrete design remains, i.e. until there are no more 'abstract' (and hence difficult to execute) language structures remaining. The resulting design may then be transformed into an implementation. Figure 2 depicts how the refinement method works.

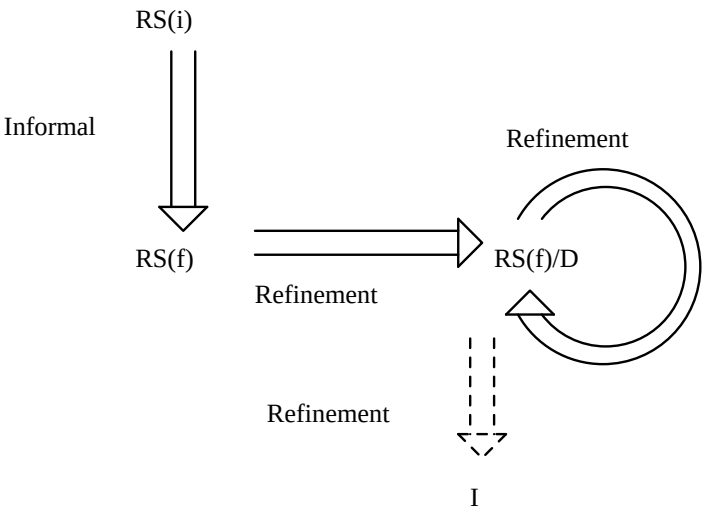

Figure 2: The Refinment Method

Examples of theories which implicitly or explicitly use this method are given in the following table, along with their representation languages:

Theory	$RS(f)$ structures	D structures	Implementation	Ref
Morgan	Pre/Post (predicates)	Dijkstra GCL	Dijkstra GCL	[Mor90]
SETL	FO Predicate Logic	Imper prog lng	Imper prog lng	[SETL85]

The advantages of the refinement method are:

- Single language. The user only has to become familiar with a single development language which is used throughout the method.
- No iteration. There is no iteration due to incorrect design: the refinement laws guarantee that the design is correct with respect to the $RS(f)$.
- Little Informality. There is only one informal translation process which potentially reduces the number of errors in the design with respect to the $RS(f)$.
- Simplicity. The proof obligations arising from refinement laws are often very simple.

Of course, there are also a number of disadvantages:

- Efficiency. Often the refinement method gives rise to inefficient code. The primary objective of refinement is correctness, and other, qualitative, metrics are often ignored.
- Re-use. It is difficult to envisage how pre-defined specification 'modules' may be combined to give an $RS(f)$. It is in the requirements that most of the uniqueness and diversity of a system occurs.

1.2.4 A Method for TAM

In this paper we propose to investigate the development of a refinement calculus for real-time systems. In doing so we will have to define a computational model (which in turn

defines the application area), a language syntax and semantics which are amenable to refinement, and an associated calculus.

1.3 Paper Structure

In the following sections the computational model for TAM is defined along with a timed logic in which to describe computations, the TAM language syntax is described and an informal semantics is defined. The refinement method for TAM is discussed and a refinement calculus presented. A short example is also undertaken. The appendix gives the formal semantics and definition of refinement.

2 A Computational Model

A computational model should reflect the intended target application area, and also define what we mean by terms such as 'termination', 'communication', 'concurrency' and 'functionality'. In doing so, it also defines the class of systems which we may represent and reason about in any language which supports the model.

2.1 Systems

We define a **real-time system** as a collection of possibly concurrently executing computation **agents** which communicate aysnchronously via time-stamped shared data areas called **shunts**. Systems can themselves be viewed as single agents and composed into larger systems. Systems may have timing constraints imposed at three different levels, system-wide communication deadlines, agent deadlines, (both on the agent computations and inter-agent communication) and sub-computation deadlines (within the computation of an individual agent). A system has a static configuration, i.e. all connections remain fixed throughout the lifetime of the system. Every system also has the constraint that there may be at most a finite number of shunts and agents.

At any instant in time the system can be thought of as having a unique **state**. The system state is defined by the combinations of the states of the shunts and the states of all agents; **computation** is therefore defined as any process which results in a change of system state. A corollory of these two definitions is that no computation may be instantaneous (take zero time) – a restriction which would seem intuitive.

2.2 Time

Time is global i.e. a single clock is accessible to every agent and shunt. It is both discrete and linear, and modelled naturally by the positive integers. There is a unique 'first time' instant from which we assume all systems will measure their execution release times for all agents, and an upper bound (denoted ∞) which is greater than all times.

2.3 Agents

An agent is described by a computation which may transform a private data-space and may read and write shunts during execution. The computation may be non-deterministic, and have both minimum and maximum execution times imposed. In particular, agents have the following properties:

- An agent performs computation and communication functions and must terminate; no agent may compute for an infinite amount of time (even if it is performing no

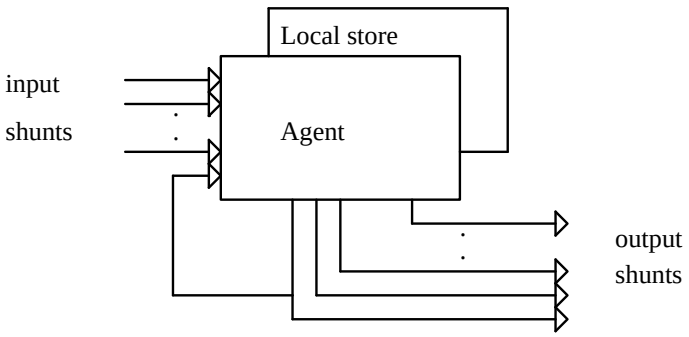

Figure 3: An Individual Agent

computation i.e. in livelock). Only an agent which performs no computation or reading or writing may be instantaneous (i.e. terminate at the same instant in time as it is released).

- An agent may start execution as a result of either a condition on the current time, or a write event occurring on a specific shunt. These two conditions model periodic and sporadic real-time tasks respectively.
- An agent may have deadlines on computation, sub-computations, and communication reads and writes. Deadlines may be **dynamic** i.e. dependent upon results of computation, or **static**. Deadlines are all considered to be **hard**, that is, there is no concept of deadline priority.
- An agent's local data-space is created when the agent starts execution with initial values which are non-deterministic; the data space is destroyed when the agent terminates. No agent may read a concurrently executing agent's private data-space.
- The size of any agent's data-space is finite.
- An agent may write to at most a finite number of shunts, and read from at most a finite number of shunts.
- An agent's output shunts are *owned* by that agent, i.e. no other agent may write to those shunts (although many other agents may read them).
- No computation may be instantaneous, i.e. no local state-space transformation or shunt reading or writing may take zero time.

We also assert that every agent has the property of **referential transparency** [Sto87] that is, if we replace an agent with one which has a different internal structure but which behaves the same on its input and output shunts (temporally and functionally), then we may interchange the two agents freely without effecting the behaviour of the complete system.

Figure 2.1 shows an individual agent with a number of input and output shunts, note that some shunts may be used for both purposes.

2.4 Shunts

Shunts are passive shared memory spaces which contain two values; the first gives the time at which the most recent write took place, and the second gives the value which was most

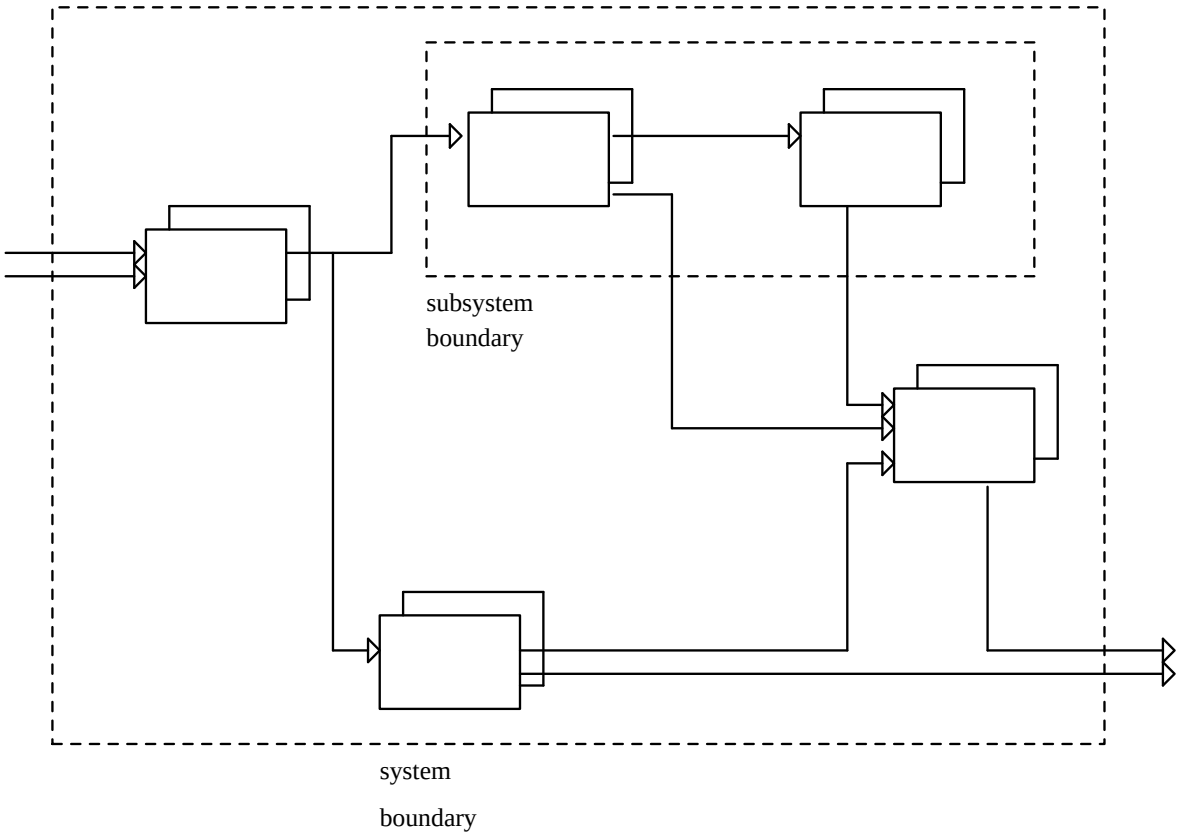

Figure 4: Example system containing 2-agent subsystem

recently written. Shunt writing is destructive, shunt reading is not. Shunts are strictly typed throughout the lifetime of the system, i.e. there is an identified set of values to which the data value of a shunt must always belong.

Shunts are loosely asynchronous; if the owning agent performs a write then the write is said to be completed when the value has been accepted by the shunt and the time-stamp is over-written by the current time. The only delay on writing is due to the communication overhead, a shunt may not refuse to accept value. When an agent performs a read then the shunt returns both the time-stamp found in the shunt, and the associated value. Again, any delay is due to communication overheads.

2.5 Partioning Subsystems

Shunt readership may be restricted to a set of agents, these can then be considered as a subsystem where shunts which are read or written by the agents within the subsystem define the subsystem's boundary. Subsystems may not overlap, i.e. if an agent is within one defined subsystem then it may not occur in another subsystem which does not wholly contain the former.

Figure 2.2 shows an example system containing a single subsystem.

We can assert, by the property of referential transparancy, that any subsystem may be replaced with a single agent, and any single agent may be replaced by a subsystem. We may therefore alternatively call an agent a subsystem, or vica-versa.

2.6 Issues Arising From Our Model

There are a number of areas in which we have reduced the scope of our computational model with respect to real-time systems in general. These areas are discussed below.

2.6.1 Termination Model

A corollory of the two constraints that assert that all agents must terminate and that there may only be a finite number of agents within any given system, is that all systems must themselves terminate. We have decided upon this constraint in order to make the analysis of systems in some sense 'easier' (which will become more apparent when we discuss the semantics of our formal language in the next chapter). However, we have to be certain that this constraint does not reduce too much the class of real-time systems which may be represented or reasoned about within our model.

Although many systems we would like to reason about do not terminate (most control systems for example), they often share the property that they in some sense repeat some finite behaviour forever. With these systems it is possible to isolate the behaviour which is repeating, and reason about it; by extrapolation we may then reason about the behaviour of the complete system.

This observation allows us to reason about a wide range of non-stop systems by representing their repeating subsystems under the appropriate time interval. In particular, systems that are *periodic* in which the repeating subsystem has a deadline no greater than the period, are temporally predictable under the subsystem with an interval equal to the period.

We will return to this issue when we present examples of system development.

2.6.2 Communication Model

The timestamps within the shunts enable the reading agents to compute according to the *freshness* of data. There are many systems within which freshness is a prime issue, for example:

- Fault-tolerant systems whose 'error-code' is executed if a heartbeat signal is not received from a monitored process within a given time interval.
- Real-time database systems where the results of queries need to be combined with freshness information.
- Wide area networks which transmit packets by sub-dividing and independently routing. Recombination of packets depend upon their sending order; information about recombination may be encoded by timestamps.

The need for timestamps in shunts is also a direct consequence of our decision to use non-destructive asynchronous communication. When an agent performs two consecutive inputs from a shunt, if it reads the same data item twice it may need to know if each value is a result of a different write, or a single write.

Having non-destructive reads enables many agents to read the same value (modelling multi-drop communication media), and also allows us to model *global state*. Asynchronous

communication is both easier to schedule, and allows us to encode very efficient communication protocols [Sim90].

It is worth noting that neither asynchronous communication, or synchronous communication is more 'primitive'. It is possible to provide protocols in one communication model which induces the other. However, to do so is inefficient, and it is therefore better to provide the model which is easier to support in the run-time system, and provide the most often used programming primitive to the user.

Single writer mechanisms avoid the need for complex resource blocking protocols. With multiple writers, there needs to be some mechanism for choosing, or ordering the access to the shared resource i.e. the shunt. We lose no generality by chosing single writers, by providing multiplex agents which read indexed shunts, multiple writing can be modelled (we show examples of this later).

The restriction of having a finite number of shunts is intuitive with respect to a physical implementation of a TAM system. It would never be possible to provide an infinite number of shunts, and so we do not allow them in our model.

2.6.3 Computation Model

The local state space of an agent is partitioned and named by **computational variables**. Each variable is of a pre-determined **type**, i.e. may only be bound to values from a given set. We are not especially interested in the mechanics of functional computation in TAM, and we assume that there is some language for expressing the transformation of the local state space. The computational model (and the TAM language) assumes that there exists some underlying functional evaluation engine which need not be understood in detail. The model only understands the act of **assignment**.

The model does however insist that computation evaluation be non-instantaneous i.e. an assignment must take at least one time unit. Input and output must also take at least one time unit as they are changing either the local state of an agent (input) or the global state (an output).

2.7 The Generality Of Our Model

The computational model provides us with a framework of great generality. Most real-time systems are viewed as a set of periodic and sporadic tasks (assuming some limitations on inter-arrival rates etc), and such systems are easily represented in the TAM model.

In particular, TAM enables the developer to describe systems as a set of agents with a number of **properties** including:

- release time and period (for periodics),
- release event and minimal inter-arrival time (for sporadics),
- a functional computation,
- hard deadlines on communications, computations, and termination,
- an input and output interface,
- upper-bounds on resource requirements (state-space and shunts).

a subset of which are often used for run-time analysis and scheduling [Tin90] [Aud91].

Other computational models for timed systems may also be *embedded* in the TAM model (the TAM model is of greater generality). For example, Tucker's analysis of synchronous concurrent algorithms [TT91] is based upon a computational model which is a subset of the TAM model; this allows us to express systems in the TAM framework such as:

- cellular automata,
- neural networks,
- systolic arrays,
- data flow,

2.8 A Logic to Describe Computations

In order to discuss behaviour of systems within our computational model, we make conservative extensions to the first-order predicate logic, and import standard ZF set theory. The extensions equate computational variables and shunts with functions mapping positive integers (time) onto value sets.

2.8.1 Timed Variables

A **timed variable** is a function mapping positive integers (representing the time) onto some value set (representing the variable type). Given a computation variable 'x' of type 'V' ($x \in [\mathbb{N} \rightarrow V]$), then we use the notation:

$$x@t$$

to denote the application of the function 'x' to the time 't', i.e.:

$$x@t \quad =_{def} \quad x(t)$$

This allows us to make assertions of the form (for example):

$$\forall\, n \in \mathbb{N}(x@(t+1) = x@t)$$

(the variable x never changes value),

$$\exists\, t \in \mathbb{N}(x@t = y@t)$$

(at some time the values of the variable x and y will be equal).

2.8.2 Shunts

Shunts are also represented by functions mapping positive integers (times) onto some value set. The value set will always be a cross-product of positive integers and some value set. Again, we use the notation:

$$s@t$$

to represent the value of the shunt 's' at some time 't'.

The value of a shunt will always be a timestamp and value pair, so we will use the tuple accessors $.ts$ and $.v$ to access a shunt value, e.g.

$$s@t \;=\; ((s@t).ts\ ,\ (s@t).v)$$

We may then make assertions of the form (for example):

$$\forall\, n \in [t, t'](n \mathrm{mod} 4 \;\Rightarrow (s@n).ts = n)$$

(the shunt s is written to once every fourth time instance in the interval $[t, t']$).

$$\forall\, n \in [t, t'](s'@(n+1).v = s@n.v)$$

(for every time instance in the interval $[t, t']$ the value of the shunt s will be the same as the value found in the shunt s' in the next time instance).

2.9 Describing System Behaviours

A system within our computational model may now be described as a conjunction of formulae in our extended logic, each conjunct describes a safety, liveness or timeliness property.

This is not however the specification method used in TAM, which uses a more constrained specification form within a wide-spectrum language syntax – this is discussed in detail in the next section.

2.10 Timed Logic and Specifications

One of the syntactic program statements in the TAM language is the *specification statement*. 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 times. 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.

It is assumed that the external configuration of a TAM system is static, and therefore the environment of an agent, i.e. the shunts which are read and written by that agent, and the local state space, are known prior to system development. Therefore, in most cases, the initial specification can be viewed as a description of a 'black-box' with a number of identified input and output channels and a local state space definition.

3 TAM Syntax

The modular abstraction mechanism in TAM is the *agent* which can be defined recursively as:

$$\mathcal{A} \;::=\; x := e \;\mid\; x \leftarrow s \;\mid\; x \rightarrow s \;\mid\; w : \Phi \;\mid\; (x : T)\mathcal{A} \;\mid\; \mathcal{A}/s : T$$

$$\mathcal{A} \mid \mathcal{A} \;\mid\; [t]\mathcal{A} : \mathcal{A} \;\mid\; \bigsqcup_{i \in I} g_i \Rightarrow \mathcal{A}_i \;\mid\; \mathcal{A} \triangleright_n^s \mathcal{A} \;\mid\; \mu\, X_n . F(X)$$

We give an informal semantics for these agent forms as follows:

$x := e$ performs an assignment of the value of the expression e (defined over the local state space of the agent) to the variable x. The expression is in the form of functional evaluation described in first-order predicate logic. The evaluation can not be instantaneous.

$x \leftarrow s$ performs an input from the shunt s into the variable x. The type of x must therefore be a natural number and value pair. The input is asynchronous and may never be instantaneous.

$x \rightarrow s$ performs an asynchronous output to the shunt s from the variable (or constant value) x. The time of the output is used as the time stamp. Output may not be instantaneous.

$w : \Phi$ is a specification statement over the variables in the local state space of the agent, and the shunts in the environment of the agent.

$(x)\mathcal{A}$ introduces the unique variable x into the local state space of the agent $\mathcal{A}$. Initially x has an undefined value (any value from the appropriate type set T). The variable is destroyed (unavailable) after the termination of $\mathcal{A}$.

$\mathcal{A}/s$ restricts access to the shunt s to the agent $\mathcal{A}$, and declares it's type as T. This operator can be seen as a 'connector' of two ports in that no communication can actually take place until a shunt is restricted.

$\mathcal{A} \mid \mathcal{A}$ executes the two agents concurrently. The two agents are released at the same time, and the concurrent composition terminates some arbitrary time after both agents terminate. We use the shorthand $\prod_{i \in I} \mathcal{A}_i$ to denote indexed concurrent composition (I is finite).

$[t]\mathcal{A} : \mathcal{A}$ assumes that the execution of the first agent will terminate sometime before t time units after release. The second agent will be released at the time that the first agent terminates. This provides a constraint on the run-time execution environment (in particular, the scheduler).

$\bigsqcup_{i \in I} g_i \Rightarrow \mathcal{A}_i$ evaluates all of the guards g_i (which are boolean expressions on local state space and shunts), and executes one of the agents corresponding to a true guard. If no guards evaluate to true, then the choice agent terminates correctly. It is also assumed that a function Δ is pre-defined which defines the evaluation time necessary for any guard g_i. (The indexing set I is always finite).

$\mathcal{A} \triangleright_n^s \mathcal{A}$ is a timeout operator. The agent waits for a write to occur on shunt s for n time units. As soon as the shunt is written to the right-hand agent is released. If after n time units no write has occurred then the left-hand agent is released.

$\mu X_n.F(X)$ represents that agent which executes the agent F and upon encountering the agent variable X will start executing F again. This recursion is bounded by the natural number variable n (where $n < \infty$).

We impose the syntactic restriction that no agent may share its' local state space with a concurrently executing agent, and only one agent may write to any given shunt.

The formal semantics for TAM are defined by extensions to Dijsktra's weakest pre-condition semantics [Dij76] and are given in the appendix.

From the primitive agents above we may derive the following useful constructs.

The agent 'skip' takes no time in which to execute, and changes no shunts or variables:

$$\text{skip} =_{def} \emptyset : \text{true}$$

A commonly used real-time statement is the delay, which assumes that no resources are needed for a minimum period. The agent 'δn' which is guaranteed to wait for n time units is derived as:

$$\delta n =_{def} (\text{skip} \triangleright_n^s \text{skip})/s$$

The agent 'abort' may not be reasoned about, it performs the task of a system annihilator:

$$\text{abort} =_{def} [0]\delta 1 : \mathit{skip}$$

A precedence constraint $\mathcal{A}; \mathcal{B}$ is defined by:

$$\mathcal{A}; \mathcal{B} =_{def} [\infty]\mathcal{A} : \mathcal{B}$$

A deadline for a single agent is defined as:

$$[t]\mathcal{A} =_{def} [t]\mathcal{A} : \text{skip}$$

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:

$$Clock(n) =_{def} \mu X_n.([1]\text{true} \rightarrow \text{time} : X)$$

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.

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':

$$Period(\mathcal{A}, p, d, n) \ = \ \mu X_n.([p](\delta p \mid [d]\mathcal{A}) : X)$$

In this example we assume deadline $\leq$ period.

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 have internal concurrency, 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:

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

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 sub-agents needs a minimum of one, three, and one time units respectively). 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 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 (we assume for simplicity that at least one such value exists):

[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

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

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

$$\begin{aligned}(n)(n := 1;\;\; \mu\, X_{\#a}.(\; \{i,n\} :& (a@t[n@t] \;\Rightarrow\; i@t' = n@t) \wedge \\ & (\neg a@t[n@t] \;\Rightarrow\; i@t' = i@t) \wedge \\ & (n@t' = (n@t) + 1);\; X))\end{aligned}$$

Or in a completely concrete agent:

$$(n)(n := 1;\;\; \mu\, X_{\#a}.(\; (a[n] \rightarrow i := n);\; n := n + 1;\; X))$$

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

More interesting, are agents of the form:

$$b \rightarrow \delta n \;\sqcup\; \neg b \rightarrow \delta m \qquad \text{(where n,m are constants)}$$

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:

$$b \rightarrow [n]\mathcal{A} \;\sqcup\; \neg b \rightarrow [m]\mathcal{A} \qquad \text{(where n,m are constants)}$$

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.

4 Refinement

4.1 The TAM Method

We propose that systems are developed in the TAM theory using the refinement method. The TAM language is wide-spectrum, and contains an abstract agent (description statement) which contains a first-order predicate logic description of the timing, functional, and communication behaviour of a reactive system. Other TAM constructs, such as the deadline statement, delay statement, and timeout statement can also be considered as partially abstract (they are amenable to execution, but specify timing constraints which are passed to the run-time system, rather than concrete statements which are directly executed).

We define the development method used in TAM explicitly by:

1. Formulate a formal requirements specification from the informal requirements specification using a single specification statement.
2. Choose any refinement law whose antecedent matches the formal requirements specification statement, and replace it with the consequent.
3. Repeat step 2 until there are no more specification statements in the description.

The refinement relation on TAM agents is considerably more complex than that used in theories such as the one developed by Mogan [Mor90]. Primarily, this is due to the complexity of the target application area (real-time systems) rather than an inherent unsuitability of refinement for reactive systems. The complement method has been used for real-time formalisms (Timed-CSP [Schn90] [Dav91]) and this has also been shown to be complex; the proof system for Timed-CSP in [Dav91] contains some forty non-trivial proof rules, and we estimate that this will be the same order of complexity as the refinement calculus for TAM.

4.2 Real-Time Refinement in TAM

There is little reference to refinement for real-time systems in the literature. It is clear that refinement in real-time systems should 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 is also intuitive to shorten deadlines and we allow this to occur whenever a shortened deadline does not make an agent impossible to execute within the time interval.

A formal definition of the refinement relation is given in the appendix, but it can be informally described as follows:

> "An agent $\mathcal{A}$ is refined by $\mathcal{B}$ whenever the environmental constraints which are necessary for $\mathcal{A}$ to achieve a given behaviour are stronger than those for $\mathcal{B}$, and, the effects of $\mathcal{B}$ on the environment are stronger (more deterministic) than those of $\mathcal{A}$"

The refinement calculus for TAM is presented in the appendix and we only discuss a representative sample here. The most important refinement law is that of specification strengthening:

$$w : \Phi \quad \sqsubseteq \quad w : \Phi' \text{ (if } \Phi' \equiv\rangle \Phi \text{ and } \Sigma(w : \Phi) = \Sigma(w : \Phi'))$$

This law asserts that if a user asks for an agent which ensures the behaviour Φ, and we provide the stronger behaviour Φ' then we are fulfilling at least the behaviour required of us. The constraint 'Σ' on specifications is defined in the appendix, and intuitively defines the minimum amount of time required for the execution of the specification in order to

ensure that the specification is feasible. The condition on Σ ensures that the refining agent does not require more resources.

We can see that given a specification:

$$\{x\} : x@t' = (x@t) + 1 \ \vee \ x@t' = (x@t) - 1$$

the requirements are that the variable x is either incremented or decremented. We may therefore refine this to:

$$\{x\} : x@t' = (x@t) - 1$$

Similarly, a specification of an input from shunt s to variable x within the time interval $t, t+5$:

$$\{x\} : \bigvee_{\sigma \in [t,t+5]} (x@t' = s@\sigma) \ \wedge \ t' \geq t + 5$$

Can be refined to the input specification which requires the input to occur between $t+1$ and $t+5$:

$$\{x\} : \bigvee_{\sigma \in [t+1,t+5]} (x@t' = s@\sigma) \ \wedge \ t' \geq t + 5$$

This rule is often used to refine specifications which are infeasible due to the limitations of our computational model into stronger, feasible, specifications (in this case the requirement that all inputs should take longer than one time unit and therefore cannot occur at the release time t).

Refinements on choice constructs must take into account the time required in which to evaluate the boolean guards, therefore the rule often found in refinement systems of the form:

$$\bigsqcup_{i \in I} g_i \Rightarrow \mathcal{A}_i \ \sqcup \ \text{true} \Rightarrow \mathcal{B} \quad \sqsubseteq \quad \mathcal{B}$$

is amended to:

$$\bigsqcup_{i \in I} g_i \Rightarrow \mathcal{A}_i \ \sqcup \ \text{true} \Rightarrow \mathcal{B} \quad \sqsubseteq \quad [\,^{\Sigma}_{i \in I} \Delta(g_i) + \Delta(true)]\delta \,^{\Sigma}_{i \in I} \Delta(g_i) + \Delta(true) : \mathcal{B}$$

In which it is assumed that the execution of the agent $\mathcal{B}$ is delayed precisely by the amount of time taken by the evaluation of all of the guards (including the guard 'true').

If we were to remove the timing considerations of the choice construct (i.e. $\Delta(g) = 0$ for all guards g), then the former refinement law would be valid:

$$\bigsqcup_{i \in I} g_i \Rightarrow \mathcal{A}_i \ \sqcup \ \text{true} \Rightarrow \mathcal{B}$$

$$\sqsubseteq \quad [0]\delta 0 : \mathcal{B}$$

$$\sqsubseteq \quad \mathcal{B}$$

where the last refinement assumes the law:

$$[0]\delta 0 : \mathcal{A} \quad \sqsubseteq \quad \mathcal{A}$$

Concurrency can be introduced by dividing a specification conjunction into concurrent specifications. This may be achieved by the following law:

$$\bigcup_{i \in I} w_i : \bigwedge_{i \in I} \Phi_i \quad = \quad \prod_{i \in I} w_i : \Phi_i$$

if w_i are disjoint

Note the requirement that the individual specifications are interference free, i.e. the frames w_i must be disjoint.

In general, the division of a specification into concurrent specifications is the only place in which concurrency is introduced into the refinement process.

5 Example - A Single-Shot Monitor

The requirements are to derive an agent which will read an input shunt s_0 within five time units of release, and if the value which was read passes a boolean test '*ok*', then no action is taken, if however the value read does not pass the test then the constant value '*alarm*' is written to the output shunt s_1. In the following refinement, the proof numbers refer to the laws defined in appendix B.1.

We may formalise the requirements by the specification statement:

Spec1 $=_{def}$

$$\begin{aligned}\{s_1\} : \exists\, x(\exists\, m(m \in [t, min(t+5, t')] \\ \wedge \bigvee_{\sigma \in [t, m']} (x@m = s_0@\sigma) \ \wedge \ stable(s_1, t, m) \ \wedge \ m \geq t+1 \\ \wedge ((\neg ok(x@m) \ \wedge \ write(s_1, alarm, m, t')) \\ \quad \vee (ok(x@m) \ \wedge \ stable(s_1, m, t'))) \\ \wedge \ t' \geq m + 5))\end{aligned}$$

It is interesting to consider how we arrived at this specification statement from the initial requirements. It is perhaps more intuitive to start with a conjunction of properties directly synthesised from the informal English. We might propose:

Property 1: $(\bigwedge_{\sigma \in [t, t+5]} \neg ok(s_0@\sigma)) \ \Rightarrow \ write(s_1, alarm, t+5, t')$

"if none of the values in s_0 are 'ok' for the first five time units then the alarm must be written to s_1"

Property 2: $t' \geq t + 6$

"there must be five units of time available in which to perform the reading, and one time unit in which to either write if necessary".

If we take a conjunction of the two properties then we may perhaps be able to claim that we have a more intuitive specification of the user's requirements (it is still under-specified because we have not said what will happen if the shunt s_0 contains both 'ok' and not 'ok' values – this is because the user didn't specify what should happen in this case).

We can now see that:

$$\{s_1\} : property1 \ \wedge \ property2 \quad \sqsubseteq \quad Spec1 \quad \text{(by 12.6)}$$

In refining to $Spec1$ we are making a number of design decisions, but none of which contradict the informal specification.

We may now continue with the refinement process.

$$Spec1 \quad \sqsubseteq$$

$$\begin{aligned}(x)\{s_1, x\} : \exists\, m(m \in [t, min(t+5, t')] \\ \wedge \ \bigvee_{\sigma \in [t, m']} (x@m = s_0@\sigma) \ \wedge \ stable(s_1, t, m) \ \wedge \ m \geq t+1 \\ \wedge \ ((\neg ok(x@m) \ \wedge \ write(s_1, alarm, m, t')) \\ \quad \vee \ (ok(x@m) \ \wedge \ stable(s_1, m, t'))) \\ \wedge \ t' \geq m + 5) \qquad \text{(by 9.4)}\end{aligned}$$

$$\sqsubseteq$$

$$\begin{aligned}(x)([5](\{s_1, x\}: \bigvee_{\sigma \in [t, t']} (x@t' = s_0@\sigma) \ \wedge \ stable(s_1, t, t') \ \wedge \ t' \geq t+1) \\ : \ \{s_1, x\} : ((\neg ok(x@t) \ \wedge \ write(s_1, alarm, t, t')) \\ \vee \ (ok(x@t) \ \wedge \ stable(s_1, t, t'))) \\ \wedge t' \geq t + 5 \ ) \qquad \text{(by 4.5)}\end{aligned}$$

We now split the sequence into the LHS and RHS.

$$\{s_1, x\} : \bigvee_{\sigma \in [t, t']} (x@t' = s_0@\sigma) \ \wedge \ stable(s_1, t, t') \ \wedge \ t' \geq t+1$$

$$\sqsubseteq \quad \{x\} : \bigvee_{\sigma \in [t, t']} (x@t' = s_0@\sigma) \ \wedge \ t' \geq t+1 \qquad \text{(by 12.3)}$$

$$\sqsubseteq \quad \{x\} : \bigvee_{\sigma \in [t+1, t']} (x@t' = s_0@\sigma) \ \wedge \ t' \geq t+1 \qquad \text{(by 12.6 and 12.4)}$$

$$\sqsubseteq \quad x \leftarrow s_0 \qquad \text{(by 2.2)}$$

$$\begin{aligned}\{s_1, x\} : ((\neg ok(x@t) \ \wedge \ write(s_1, alarm, t, t')) \\ \vee \ (ok(x@t) \ \wedge \ stable(s_1, t, t'))) \\ \wedge t' \geq t + 5\end{aligned}$$

$$\sqsubseteq$$

$$\begin{aligned}\{s_1, x\} : ((\neg ok(x@t) \ \wedge \ write(s_1, alarm, t+1, t') \ \wedge \ t' \geq (t+1)+4) \\ \vee \ (ok(x@t) \ \wedge \ stable(s_1, t+1, t') \ \wedge \ t' \geq (t+1)+4)) \\ \wedge \ all\_stable(\{s_1\}, t, t+1) \qquad \text{(by 12.6)}\end{aligned}$$

$$\sqsubseteq$$

$$\begin{aligned}&\neg ok(x) \Rightarrow (\{s_1\} : write(s_1, alarm, t, t') \ \mid \ \emptyset : t' \geq t+4) \\ &ok(x) \Rightarrow (\{s_1\} : stable(s, t, t') \ \mid \ \emptyset : t' \geq t+4) \\ &\quad \text{(by 7.9, 5.5,12.6, 12.7, assuming } \Delta(ok(x)) + \Delta(\neg ok(x)) = 1)\end{aligned}$$

$\sqsubseteq$

$\neg ok(x) \Rightarrow (alarm \rightarrow s_1) \mid \delta 4)$
$ok(x) \Rightarrow \delta 4$
(by 3.2,11.2,12.3,5.3).

We may now place our sequence halves back into the original agent to give:

$(x)([5](x \leftarrow s_0) : ok(x) \Rightarrow \delta 4$
$\neg ok(x) \Rightarrow (alarm \rightarrow s_1 \mid \delta 4))$

6 Conclusion

We have presented a wide-spectrum language and refinement calculus for the development of real-time systems. It is clear that developing such systems will remain a complex and time consuming task independent of the method employed, but we have demonstrated that a refinement method is both feasible and useful for hard real-time systems.

Our development method does not address issues such as schedulability and process allocation. Work by Hooman [Hoo91] and Joseph [MJ90] implies that useful results can be obtained by modelling resource issues in the development language – we will investigate this further in the near future. However, TAM provides a language in which real-time specifications can be refined to task sets and profiles (given by the deadlines embedded in the concrete code), and interfaces with existing scheduling theory and run-time execution environments.

Perhaps the most important factor in formal development is that of complexity. The refinement calculus of TAM has complex proof obligations, in the future, an automated theorem checker such as HOL [Gor88] may be used.

7 References

[Aud91] N.C. Audsley, "Optimal Priority Assignment and Feasibility of Static Priority Tasks with Arbitrary Start Times", University of York Technical Report YCS-164. 1991.

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

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

[Gor88] M. J. Gordon, "Mechanizing Programming Logics in Higher-Order Logic", University of Cambridge Technical Report 145. September 1988.

[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.

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

[MJ90] A. Moitra, M. Joseph, "Implementing Real-Time Systems by Transformation", in "Real-Time Systems, Theory and Practice", H.S.M. Zedan (ed). Elsevier Science Publishers B.V. 1990.

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

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

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

[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.

[Sim90] H.R. Simpson, "Four-slot Fully Asynchronous Communication Mechanism", IEE Proceedings, Vol. 137, Pt.E, No.1, Jan 1990.

[Sto87] J.E. Stoy, "Denotational Semantics", MIT Press. 1987.

[SZ91] D. Scholefield, H.S.M. Zedan, "The Temporal Agent Model: Theory and Practice", University of York Technical Report YCS-163. 1991.

[SZ92] D. Scholefield, H.S.M. Zedan, "TAM: A Formal Framework for the Development of Distributed Real-Time Systems", in "Formal Techniques in Real-Time and Fault-Tolerant Systems", Nijmegen LNCS 571. 1992.

[Tin92] K. Tindell, A. Burns, A. Wellings, "Allocating Real-Time Tasks (An NP-Hard Problem made Easy)", Real-Time Systems Journal, (to appear).

[TM89] C. Tofts, F. Moller. "A Temporal Calculus Of Communicating Systems". Edinburgh University, Technical Report LFCS-89-104. 1989.

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

[TT91] B.C. Thompson, J.R. Tucker, "Equational Specification of Synchronous Concurrent Algorithms and Architectures", University of Wales, University College of Swansea Technical Report CSR-9-91. 1991.

A Definitions

The following definitions are used by the predicate transformer semantics of TAM.

A.1 Substitutions

We assume a semantics of predicate logic in which interpretations are functions binding constants to values (and functions to definitions). In our specification statements timed variables and shunts always occur as free constants and therefore have a binding in the interpretation. In such a semantics we may define point-wise substitution as follows:

$$\text{Definition: (Point Substitution)} \left| \begin{array}{l} I \models \Phi[x/y] \text{ for free variables } x,y, \text{ exactly when} \\ \\ I \oplus \{y \mapsto I(x)\} \models \Phi \end{array} \right|$$

(where $\oplus$ is function override). We define a specific multiple substitution of computation variables referenced at one time with variables at a second time.

$$\text{Definition: (Timed Substitution)} \left| \begin{array}{l} I \models [\![\Phi]\!]^{u}_{\delta} \text{ exactly when} \\ \\ I \models \Phi[x\tilde{@}\delta/x\tilde{@}u] \\ \\ \text{(for any timed variable } x@t) \end{array} \right|$$

We allow the quantification of a variable or shunt at a specific time by:

$$\text{Definition: (Timed Quantification)} \left| \begin{array}{l} I \models \forall\, x@t(\phi) \text{ iff} \\ \\ I \models \forall\, X(\phi[X/x@t]) \end{array} \right|$$

The quantification can be extended over sets of variables and shunts in the usual way.

We define an untimed quantifications for variables and shunts as follows:

$$\text{Definition: (Untimed Quantification)} \left| \begin{array}{l} I \models \forall\, x(\phi) \text{ iff} \\ \\ I \models \forall\, X(\phi[X/x\tilde{@}t]) \text{ (for any time } t) \end{array} \right|$$

Discharge is a meta-level function which denotes the minimum formula which needs to hold to ensure that the communication predicate entails the functional requirement. The partial ordering which defines 'least' is that of entailment.

$$\text{Definition: (Discharge)} \left| \begin{array}{l} \mathit{Discharge}(\phi, \psi) \text{ is the } \mathit{least} \\ \text{formula } \Xi \text{ such that:} \\ (\psi \wedge \Xi) \equiv\rangle \phi \end{array} \right|$$

B Refinement

We define refinement informally as:

> An agent $\mathcal{A}$ is refined by $\mathcal{B}$ (denoted $\mathcal{A} \sqsubseteq \mathcal{B}$) in those cases where $\mathcal{B}$ needs a weaker constraint on the environment than $\mathcal{A}$ in order to achieve at least as powerful a behaviour. The constraints include the minimum amount of time required, and the value of shunts during execution of $\mathcal{A}$ and $\mathcal{B}$.

We may formally define refinement within the weakest pre-condition semantics as follows:

Definition 12: (Refinement)	$\mathcal{A} \sqsubseteq \mathcal{B}$ iff for any interval $[\delta, u]$, post-condition ϕ, communication predicate ψ, then: $wp^u_\delta(\mathcal{A}, \phi, \psi) : w.f \;\; \equiv\!\rangle \;\; wp^u_\delta(\mathcal{B}, \phi, \psi) : w.f \quad (i)$ $wp^u_\delta(\mathcal{B}, \phi, \psi) : w.c \;\; \equiv\!\rangle \;\; wp^u_\delta(\mathcal{A}, \phi, \psi) : w.c \quad (ii)$

We define the refinement calculus by the following theorems:

B.1 Refinement Laws

We denote bi-refinement, i.e. $\mathcal{A} \sqsubseteq \mathcal{B} \;\; \wedge \;\; \mathcal{B} \sqsubseteq \mathcal{A}$ as $\mathcal{A} = \mathcal{B}$.

Law 1. (Assignment)

1.1 $x := x \;\; = \;\; \delta 1$

1.2 $\{x\} : x@t' = e@t \;\wedge\; t' \geq t+1 \;\; = \;\; x := e$

Law 2. (Input)

2.1 $(x)(x \leftarrow s) \;\; = \;\; \delta 1$

2.2 $\{x\} : \bigvee_{\sigma \in [t+1, t']} s@\sigma = x@t' \;\; = \;\; x \leftarrow s$

Law 3. (Output)

3.1 $(x \rightarrow s)/s \;\; = \;\; \delta 1$

3.2 $\{s\} : write(s, x@t, t, t') \;\; = \;\; x \rightarrow s$

Law 4. (Sequence)

4.1 $[n]abort : \mathcal{A} \;\; = \;\; abort$

4.2 $[n]\mathcal{A} : abort \quad = \quad abort$

4.3 $[0]skip : \mathcal{A} \quad = \quad \mathcal{A}$

4.4 $;\ _I[n_i](x := e_i) : A \quad = \quad [\sum_{i \in I} n_i](x := \tilde{e}_i \ \mid \ \delta \# I) : \mathcal{A}$

if $\forall\, i \in I(n_i > 0)$

(where $\tilde{e}_i$ is the ordered composition of expressions e_i)

4.5 $w : \Phi \quad \sqsubseteq \quad [n](w : \Phi_0) : (w : \Phi_1)$

(if $\exists\, m(m \in I \ \wedge \ \Phi_0[m/t'] \ \wedge \ \Phi_1[m/t]) \ \equiv \ \Phi$

if $I = [t, min(t+n, t')]$ and $\Sigma(w : \Phi_0) + \Sigma(w : \Phi_1) \ = \ \Sigma(w : \Phi)$

Law 5. (Concurrency)

5.1 $\mathcal{A} \mid \mathcal{B} \quad = \quad \mathcal{B} \mid \mathcal{A}$

5.2 $\mathcal{A} \mid (\mathcal{B} \mid \mathcal{C}) \quad = \quad (\mathcal{A} \mid \mathcal{B}) \mid \mathcal{C}$

5.3 $\mathcal{A} \mid skip \quad = \quad skip$

5.4 $\mathcal{A} \mid abort \quad = \quad abort$

5.5 $\bigcup_{i \in I} w_i : \bigwedge_{i \in I} \Phi_i \quad = \quad \prod_{i \in I} w_i : \Phi_i$

if w_i are disjoint and $w_i : \Phi_i$ are interference free

Law 6. (Timeout)

6.1 $(\mathcal{A} \triangleright^s_n \mathcal{B}) \mid (\mathcal{C} \triangleright^s_n \mathcal{D}) \quad = \quad (\mathcal{A} \mid \mathcal{C}) \triangleright^s_n (\mathcal{B} \mid \mathcal{D})$

6.2 $\mathcal{A} \triangleright^s_{n+1} \mathcal{B} \quad = \quad (\mathcal{A} \triangleright^s_n \mathcal{B}) \triangleright^s_1 \mathcal{B}$

6.3 $\mathcal{A} \triangleright^s_n (\mathcal{C} \triangleright^s_m \mathcal{B}) \quad = \quad \mathcal{A} \triangleright^s_n \mathcal{B}$

6.4 $w : \Phi \quad \sqsubseteq \quad w : \Phi_0 \triangleright^s_n w : \Phi_1$

if $\Phi \ \equiv \ t' \geq t + n \ \wedge$

$((stable(s, t, t+n) \ \wedge \ all\_stable(w, t, t+n) \ \wedge \ \Phi_0[t+n/t]) \vee$

$(\bigvee_{\sigma \in [t, t+n]} (s@\sigma.1 = \sigma \wedge stable(s, t, \sigma \ominus 1) \wedge all\_stable(w, t, \sigma) \wedge \Phi_1[\sigma/t]))$

Law 7. (Guards)

7.1 $b_1 \Rightarrow \mathcal{A} \ \sqcup \ b_2 \Rightarrow \mathcal{B} \quad = \quad b_2 \Rightarrow \mathcal{B} \ \sqcup \ b_1 \Rightarrow \mathcal{A}$

7.2 $true \Rightarrow \mathcal{A} \quad = \quad [\Delta(true)]\delta\Delta(true) : \mathcal{A}$

7.3 $false \Rightarrow \mathcal{A} \;=\; \delta\Delta(false)$

7.4 $b_1 \Rightarrow \mathcal{A} \;\sqcup\; b_2 \Rightarrow \mathcal{A} \;=\; (b_1 \vee b_2) \Rightarrow \mathcal{A}$

(if $\Delta(b_1) + \Delta(b_2) = \Delta(b_1 \vee b_2)$

7.5 $b_1 \Rightarrow (b_2 \Rightarrow \mathcal{A}) \;=\; (b_1 \wedge b_2) \Rightarrow \mathcal{A}$

(if $\Delta(b_1) + \Delta(b_2) = \Delta(b_1 \wedge b_2)$

7.6 $[n](\bigsqcup_{i\in I} g_i \Rightarrow \mathcal{A}_i) : \mathcal{B} \;=\; \bigsqcup_{i\in I} g_i \Rightarrow [n - \overset{\Sigma}{}_{i\in I}\Delta(g_i)]\mathcal{A}_i : B$

if $\overset{\Sigma}{}_{i\in I}\Delta(g_i) \leq n$

7.7 $[n](\bigsqcup_{i\in I} g_i \Rightarrow \mathcal{A}_i) : \mathcal{B} \;=\; abort$

if $\overset{\Sigma}{}_{i\in I}\Delta(g_i) > n$

7.8 $(x)(\bigsqcup_{i\in I} g_i \Rightarrow \mathcal{A}_i) \;=\; \bigsqcup_{i\in I} g_i \Rightarrow (x)\mathcal{A}_i$

if $x \notin vars(g_i)$ for all i

7.9 $w : \Phi \;\sqsubseteq\; \bigsqcup_{i\in I} g_i \Rightarrow w_i : \Phi_i$

if $\Phi \;\equiv\; \exists\, \sigma(\sigma = \overset{\Sigma}{}_{i\in I\Delta(g_i)} \;\wedge$

$((\bigvee_{i\in I}(g_i@\delta \;\wedge\; \Phi_i[t+\sigma/t]) \vee$

$(\bigwedge_{i\in I}\neg g_i@\delta \;\wedge\; all\_stable(w, t, t+\sigma)))$

and for each i, $\Sigma(w_i : \Phi_i) \;=\; \Sigma(w : \Phi) - \overset{\Sigma}{}_{i\in I}\Delta(g_i)$

7.10 $\bigsqcup_{i\in I} g_i \Rightarrow \mathcal{A}_i \;\sqcup\; false \Rightarrow \mathcal{B} \;\sqsubseteq\; [\Delta(false)]\delta\Delta(false) : \bigsqcup_{i\in I} g_i \Rightarrow \mathcal{A}_i$

7.11 $\bigsqcup_{i\in I} g_i \Rightarrow \mathcal{A}_i \;\sqcup\; true \Rightarrow \mathcal{B} \;\sqsubseteq\; [\overset{\Sigma}{}_{i\in I}\Delta(g_i)+\Delta(true)]\delta\, \overset{\Sigma}{}_{i\in I}\Delta(g_i)+\Delta(true) : \mathcal{B}$

7.12 $\bigsqcup_{i\in I} g_i \Rightarrow \mathcal{A}_i \;\sqcup\; b_1 \Rightarrow \mathcal{B} \;\sqcup\; b_2 \Rightarrow \mathcal{B} \;\sqsubseteq\; \bigsqcup_{i\in I} g_i \Rightarrow \mathcal{A}_i \;\sqcup\; (b_1 \vee b_2) \Rightarrow \mathcal{B}$

if $\Delta(b_1) + \Delta(b_2) \;=\; \Delta(b_1 \vee b_2)$

Law 8. (Recursion)

8.1 $\mu\, X_0.F(X) \;=\; skip$

8.2 $\mu\, X_1.F(X) \;=\; F[skip/X]$

8.3 $\mu\, X_{n+1}.F(X) \;=\; F(\mu\, X_n.F(X))$

8.4 $w : \Phi \;\sqsubseteq\; \mu\, X_n.[p](w : \Phi') : X$

if $\Phi \;\equiv\; \exists\, \tilde{m_i}((\bigwedge_{i\in[1,n]}\Phi'[m_i/t'][m_{i-1}/t])[t/m_n] \wedge$

$$\forall\, i \in [1,n](m_i \leq m_{i-1} + p) \ \wedge \ t' \geq m_n \ \wedge \ \mathit{all\_stable}(w, m_n, t'))$$

Law 9. (Variable Restriction)

9.1 $(x)\mathcal{A} \ = \ \mathcal{A}$ (if x not in $\mathcal{A}$)

9.2 $(x)((y)\mathcal{A}) \ = \ (y)((x)\mathcal{A})$

9.3 $(x)(\mathcal{A} \mid \mathcal{B}) \ \sqsubseteq \ (x)\mathcal{A} \mid (x)\mathcal{B}$

9.4 $w : \exists\, x(\Phi) \ = \ (x)(w \cup \{x\} : \Phi)$ (if $x \notin w$)

Law 10. (Shunt Restriction)

10.1 $\mathcal{A}/s \ = \ \mathcal{A}$ if s not in environment $\mathcal{A}$

10.2 $(\mathcal{A}/s)/s' \ = \ (\mathcal{A}/s')/s$

10.3 $w : \exists\, s(\Phi) \ = \ (w : \Phi)/s$

Law 11. (General)

11.1 $\mathit{skip} \ = \ \delta 0$

11.2 $\emptyset : t' \geq t + n \ = \ \delta n$

Law 12. (Specification)

12.1 $w : \Phi \ = \ w : \Phi[x@t/x@t']$ (if $x \notin w$)

12.2 $w : \Phi \ = \ w : \Phi[x@t'/x@t]$ (if $x \notin w$)

12.3 $w \cup \{s\} : \Phi \ \wedge \ \mathit{stable}(s,t,t') \ = \ w - s : \Phi$ (if s not in Φ)

12.4 $w : \Phi \ = \ w : \Phi \ \wedge \ t' \geq t + n$ (if $\Sigma(w : \Phi) = n$)

12.5 $w : \Phi_0 \vee \Phi_1 \ \sqsubseteq \ w : \Phi_0$ (if $\Sigma(w : \Phi_0) = \Sigma(w : \Phi_1 \vee \Phi_0)$)

12.6 $w : \Phi \ \sqsubseteq \ w : \Phi'$ (if $\Phi' \equiv\rangle \Phi$ and $\Sigma(w : \Phi) = \Sigma(w : \Phi')$)

12.7 $w \cup \{x\} : \Phi \ \wedge \ \mathit{stable}(x,t,t') \ = \ w : \Phi$ (if $x \notin \Phi$)

C 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).

C.1 Post-Conditions, Pre-Conditions and Environments

A post-condition is a first-order predicate logic formula in which the only variables occurring free are timed variables referred to at the release and termination times of the agent, and shunts referred to 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.

C.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:

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

C.3 Supporting Concurrency

We pre-define a number of useful predicates for specifying the behaviour of shunts. In particular, we model shunt writing, reading and stability by the following predicates:

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

$$\text{Definition: (Stability)} \left| \begin{array}{l} stable(s, \delta, u) \ =_{def} \\ \\ \bigwedge_{\sigma \in [\delta, u]} s@\sigma = s@(\sigma \ominus 1) \end{array} \right|$$

(where $0 \ominus 1 = 0$ and $n + 1 \ominus 1 = n$).

The predicate *stable* is extended over sets of shunts:

$$\text{Definition: (Set Stability)} \left| \begin{array}{l} all\_stable(S, \delta, u) \ =_{def} \\ \\ \bigwedge_{s \in S} stable(s, \delta, u) \end{array} \right|$$

We also define the predicate stable (overloaded) for variables x:

$$\text{Definition: (Stability)} \left| \ stable(x, \delta, u) \ =_{def} \ x@\delta = x@u \ \right|$$

and sets of variables:

$$\text{Definition: (Set Stability)} \left| \begin{array}{l} all\_stable(X, \delta, u) \ =_{def} \\ \\ \bigwedge_{x \in X} stable(x, \delta, u) \end{array} \right|$$

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:

$$\text{Definition: (Write)} \left| \begin{array}{l} write(v, s, \delta, u) \ =_{def} \\ \\ \bigvee_{\sigma \in [\delta+1, u]} (s@\sigma = (\sigma, v) \ \wedge \ stable(s, \delta, \sigma - 1) \ \wedge \ stable(s, \sigma + 1, u)) \end{array} \right|$$

An important property of specifications is that of $\Sigma - feasibility$. The Σ-feasibility number defines the minimum amount of time needed in order to ensure that the specification describes a behaviour which is not logically inconsistent.

We define the sigma feasibility number as:

Definition:(Σ-feasibility)	The Σ-feasibility number of a specification, denoted $\Sigma(w:\Phi)$, is defined as the smallest positive integer n such that: $\models \ (t' \geq t+n) \Rightarrow \exists\, w(\Phi)$

Example.

We specify a behaviour which increments the value of a variable. The specification is given as:

$$\{x\} : x@t' = x@t + 1 \tag{i}$$

Theorem: $\Sigma(w : \{x\} : x@t' = x@t + 1) = 1$

Proof:

$$\models \quad t' \geq t + 0 \Rightarrow \exists\, x(x@t' = x@t + 1)$$

is not valid because these does not exist an x such that when $t = t'$ we have $x@t' = x@t+1$. However, for:

$$\models \quad t' \geq t + 1 \Rightarrow \exists\, x(x@t' = x@t + 1)$$

there always is an x for which $(x@t) + 1 = x@t$.

In general the Σ-feasibility number of a specification is easy to determine. In addition, we can assert the following theorems:

$\Sigma - f(1) \quad \Sigma(w : \text{true}) \ = \ 0$

$\Sigma - f(2) \quad \Sigma(w : \text{false}) \ = \ \infty$

Note in particular, the Σ-feasibility number of the specification $\{s\} : \mathit{write}(s, v, t, t')$ for any interval $[t, t']$ is one i.e:

Theorem: $\Sigma(\{s\} : \mathit{write}(s, v, t, t')) = 1$

Proof.

By definition of *stable* and *write* the indexed disjunction over $[t+1, t']$ means that $t' \geq t+1$.

C.4 Transformer Definitions

In the following predicate transformers, the notation $[\![\phi]\!]^u_\delta$ 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:

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

(where s is a shunt)

C.4.1 Assignment

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

$$(\delta < u) \;\wedge\; [\![\phi[e@\delta / x@u]]\!]^u_\delta,\; (\delta < u) \;\wedge\; [\![\psi[e@\delta / x@u]]\!]^u_\delta \;\wedge\; all\_stable(w, \delta, u)$$

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.

C.4.2 Deadline Sequence

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

$$(\bigvee_{\sigma \in [\delta, n]} wp^\sigma_\delta(\mathcal{A}, true, true) : w.f) \wedge$$

$$(\bigwedge_{\sigma \in [\delta, n]} (wp^\sigma_\delta(\mathcal{A}, true, true) : w.f \;\Rightarrow\; wp^\sigma_\delta(\mathcal{A}, wp^u_\sigma(\mathcal{B}, \phi, \psi) : w) : w.f) \;,$$

$$(\bigvee_{\sigma \in [\delta, n]} (wp^\sigma_\delta(\mathcal{A}, true, true) : w.f \;\wedge\; wp^\sigma_\delta(\mathcal{A}, wp^u_\sigma(\mathcal{B}, \phi, \psi) : w) : w.f)$$

(where $n = minimum(\delta + t, u)$). The deadline sequence expects the agent $\mathcal{A}$ to terminate before n time units after δ, and releases $\mathcal{B}$ as soon as this occurs.

C.4.3 Specification

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

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

$$\Sigma(w : \Phi) + \delta \leq u \;\wedge$$

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

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

C.4.4 Recursion

$$wp^u_\delta(\mu\, X_v.\mathcal{A}, \phi, \psi) : w \;=\; wp^u_\delta(\mathcal{A}^v(\text{skip}), \phi, \psi) : w$$

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).

The recursion operator offers a bounded iteration of the agent $\mathcal{A}$ before terminating correctly.

C.4.5 Guards

$$wp^u_\delta(\bigsqcup_{i \in I} g_i \rightarrow \mathcal{A}_i, \phi : \psi) : w \;=$$

$$(\sum_{i \in I} \Delta(g_i)) + \delta \leq u \;\wedge$$

$$((\bigwedge_{i \in I} \neg g_i@\delta \;\wedge\; [\![\phi]\!]^u_\delta) \vee$$

$$((\bigvee_{i\in I} g_i@\delta \ \wedge \ \bigwedge_{\subset\in I}(g_i@\delta \Rightarrow [\![wp^{u}_{\sum_{i\in I}\Delta(g_i)}(\mathcal{A}_i,\phi,\psi) : w.f]\!]_{\delta}^{\sum_{i\in I}\Delta(g_i)})))) \ ,$$

$$(\sum_{i\in I}\Delta(g_i)) + \delta \leq u \ \wedge$$

$$((\bigwedge_{i\in I}\neg g_i@\delta \wedge [\![\psi]\!]^{u}_{\delta} \wedge \mathit{all\_stable}(w,\delta,u)\ )) \vee \quad (\bigvee_{\subset\in I} g_i@\delta \wedge \mathit{all\_stable}(w,\delta,\sum_{i\in I}\Delta(g_i)) \wedge$$

$$(\bigvee_{i\in I} g_i@\delta \ \wedge \ [\![wp^{u}_{\sum_{i\in I}\Delta(g_i)}(\mathcal{A}_i,\phi,\psi) : w.c]\!]_{\delta}^{\sum_{i\in I}\Delta(g_i)}))))\ ,$$

The function Δ defines the amount of time which will be taken for the evaluation of each guard. The worst case time is assumed, and the RT-EE will pad to the end of this period before releasing a guarded agent. The guards are all evaluated with the values of the variables at time δ. Multiple true guards results in non-determinism. If there are no true guards then the agent will delay for the period of guard evaluation and then terminate correctly.

C.4.6 Output

$$wp^{u}_{\delta}(x \rightarrow s,\phi,\psi) : w \ =$$

$$\delta < u \ \wedge \ [\![\phi]\!]^{u}_{\delta},\ \delta < u \ \wedge \ \mathit{write}(x@\delta, s, \delta, u) \ \wedge \ \mathit{all\_stable}(w - s, \delta, u) \ \wedge \ [\![\psi]\!]^{u}_{\delta}$$

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').

C.4.7 Input

$$wp^{u}_{\delta}(x \leftarrow s,\phi,\psi) : w \ =$$

$$\delta < u \wedge [\![\bigwedge_{\sigma\in[\delta+1,u]}\phi[s@\sigma/x@u]]\!]^{u}_{\delta}, \quad \delta < u \wedge \mathit{all\_stable}(w,\delta,u) \wedge [\![\bigvee_{\sigma\in[\delta+1,u]}\psi[s@\sigma/x@u]]\!]^{u}_{\delta}$$

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.

C.4.8 Concurrency

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

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

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

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 set X^{*} must be equivalent to the original, i.e:

$$\exists X^{*}(\phi_{\mathcal{A}} \ \wedge \ \phi_{\mathcal{B}}) \ \equiv \ \phi$$

$$\exists X^{*}(\psi_{\mathcal{A}} \ \wedge \ \psi_{\mathcal{B}}) \ \equiv \ \psi$$

$$\mathit{vars}(\phi_{a}) \ \cup \ \mathit{vars}(\mathcal{B}) \ = \ \emptyset$$

$$vars(\phi_b) \cup vars(\mathcal{A}) = \emptyset$$

$$vars(\psi_a) \cup vars(\mathcal{B}) = \emptyset$$

$$vars(\psi_b) \cup vars(\mathcal{A}) = \emptyset$$

C.4.9 Restriction

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

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

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

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:

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

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

C.4.10 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.

C.4.11 Timeout

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

$$\delta + t \leq u \ \wedge$$

$$(stable(s, \delta, \delta + t) \wedge (\llbracket wp^u_{\delta+t}(\mathcal{A}, \phi, \psi) : w.f \rrbracket^{\delta+t}_\delta) \vee$$

$$(\bigvee_{\sigma \in [\delta, \delta+t]} (s@\sigma.1 = \sigma \wedge \neg stable(s, \delta, \sigma) \wedge \llbracket wp^u_\sigma(\mathcal{B}, \phi, \psi) : w.f \rrbracket^\sigma_\delta)),$$

$$\delta + t \leq u \ \wedge$$

$$(stable(s, \delta, \delta + t) \Rightarrow (\llbracket wp^u_{\delta+t}(\mathcal{A}, \phi, \psi) : w.c \rrbracket^{\delta+t}_\delta \wedge all\_stable(w, \delta, \delta + t))) \wedge$$

$$(\bigwedge_{\sigma \in [\delta, \delta+t]} ((s@\sigma.1 = \sigma \wedge \neg stable(s, \delta, \sigma)) \Rightarrow$$

$$(\llbracket wp^u_\sigma(\mathcal{B}, \phi, \psi) : w.c \rrbracket^\sigma_\delta \wedge all\_stable(w, \delta, \sigma)))$$

D Semantic Theorems

D.1 Monotonicity

We assert that given an entailment ordering between two post-conditions, and between two communication predicates, then the predicate transformers respect those orderings, i.e:

WP-Theorem 1:(WP-Monotonicity)

If $\phi \equiv\!\rangle \phi'$ and $\psi' \equiv\!\rangle \psi'$ then
for any agent $\mathcal{A}$, interval $[\delta, u]$:

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

$$wp^u_\delta(\mathcal{A}, \phi', \psi') : w.c \equiv\!\rangle wp^u_\delta(\mathcal{A}, \phi, \psi) : w.c \quad (ii)$$

This theorem is used extensively in the proof of monotonicity of the refinement calculus.

Proof of WP-Theorem 1. The proof is in 11 parts (one for each language operator). The proof is available from the authors.

D.2 Distribution of Conjunction

The proof obligations arising in the soundness of refinement calculus laws can be greatly simplified if we can subdivide weakest pre-condition calculations into separate conjunctions. This is justified by the following theorem:

WP-Theorem 2:(WP-Disjunction)

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

$$wp^u_\delta(\mathcal{A}, \phi, \psi) : w.f \ \wedge \ wp^u_\delta(\mathcal{A}, \phi', \psi') : w.f$$

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

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

The proof of this theorem is in 11 parts (one for each language constructor. It is available from the authors.

E Extra Language Operators for TAM: The Watchdog Timer

In this section we present three new language operators for TAM as an exercise in user extentions. For each operator we provide an informal and formal semantics, examples, and equational theory.

E.1 Multiple Assignments

E.1.1 Syntax

$\tilde{x_i} \;:=\; \tilde{e_i}$

to denote the simultaneous assignment of e_j to x_j for each indexed assignment and expression. The assignments occur with inner expressions binding to inner variables. The statement is well-formed when the type of the expressions match the types of the appropriate variables, and the variables x_i are disjoint.

E.1.2 Examples

$x, y \;:=\; y, x$

Performs the transfer of the variables x and y, i.e. if t is the same type as x and y, then for any n:

$$[3+n](x, y := y, x \;\mid\; \delta 3) \;=\; [3+n]((t)(t := y;\;\; y := x;\;\; x := t))$$

$x, y \;:=\; y+1, x-1$

Performs the transfer of the decrement of x to y and the increment of y to x.

E.1.3 Semantics

$wp^u_\delta(\tilde{x_i} \;:=\; \tilde{e_i}, \phi, \psi) : w \;=_{def}$

$$\delta < u \;\wedge\; [\![\phi[\tilde{x_i}/\tilde{e_i}]]\!]^u_\delta \;\;,\;\; \delta < u \;\wedge\; [\![\psi[\tilde{x_i}/\tilde{e_i}]]\!]^u_\delta \;\wedge\; \mathit{all\_stable}(w, \delta, u)$$

E.1.4 Equational Theory

1. $\tilde{x_i} \;:=\; \tilde{e_i} \;=\; \prod_{i \in I} x_i \;:=\; e_i$ if no x_i appears in e_j (i $\neq$ j)
2. $\tilde{x_i}, y \;:=\; y, \tilde{e_i} \;\sqsubseteq\; \tilde{x_i} \;:=\; \tilde{e_i}$
3. $\{\tilde{x_i}\} : \bigwedge_{i \in I} x_i @ t' = e_i @ t \;\sqsubseteq\; \tilde{x_i} \;:=\; \tilde{e_i}$

E.2 Current Time

E.2.1 Syntax

$t \downarrow$

assigns the current time to the natural number variable t. The current time is chosen from sometime between the release and termination times of the agent $t \downarrow$, and can only be reliably used as a minimum time bound.

E.2.2 Examples

$(t : \mathit{Time})(x : \mathit{Time})([1]t \downarrow : \mu\, X_{50}.[1]x \downarrow : (x < t + n) \Rightarrow X)$

will wait for a minimum of n time units (or 50 iterations, which ever is the shortest) before terminating.

$[d](\mathcal{A} : [1]t_{\mathcal{A}} \downarrow) \;\mid\; [d](\mathcal{B} : [1]t_{\mathcal{B}} \downarrow)$

will record the times at which the agents $\mathcal{A}$ and $\mathcal{B}$ terminate.

E.2.3 Semantics

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

$$\delta < u \;\wedge [\![\bigwedge_{\sigma \in [\delta+1, u]} \phi[\sigma/t@u]]\!]^u_\delta \;\;,\;\; \delta < u \;\wedge [\![\bigvee_{\sigma \in [\delta+1, u]} \psi[\sigma/t@u]]\!]^u_\delta \;\wedge\; all\_stable(w, \delta, u)$$

E.2.4 Equational Theory

1. $(t : Time)(t\downarrow) \;=\; \delta 1$

2. $\{x : Time\} : \bigvee_{\sigma \in [t+1, t']} x@t' = \sigma \;\sqsubseteq\; x\downarrow$

E.3 Funnel

E.3.1 Syntax

$s_i \rightsquigarrow_I s_v$

connects the shunts s_i (indexed by I) to the shunt s_v so that if any write occurs in a shunt s_j, $(j \in I)$ then it occurs at the same time in s_v. If no write occurs to any of the shunts s_i then s_v remains stable, and if two different values are written to shunts s_i and s_j then the funnel aborts (aborting any concurrently executing agents).

The intuition behind the funnel is that it allows concurrently executing agents to write to the same shunt, assuming that no conflict may occur.

E.3.2 Examples

$([1](x \leftarrow a) : x > val \Rightarrow true \rightarrow s_0 \;\mid\; [1](y \leftarrow b) : y > val \Rightarrow true \rightarrow s_1 \;\mid\; s_i \rightsquigarrow_{\{0,1\}} s_{out})/\{s_0, s_1\}$

reads the two shunts a and b and checks their values against the pre-determined limit. If either inputs exceeds the limit, then the shunt s_{out} is written to. Note that there can never be any value except *true* written to s_0 and s_1, and therefore there will never be conflict in writing to s_{out}.

Figure 1 gives a graphic presentation of the operation of the funnel agent.

E.3.3 Semantics

$wp^u_\delta(s_i \rightsquigarrow_I s_v, \phi, \psi) : w \;=_{def}$

$$[\![\phi]\!]^u_\delta \;\;,\;\; \bigwedge_{\sigma \in [\delta, u]} (\bigwedge_{i \in I} s_i@\sigma.ts \neq \sigma \;\wedge\; s_v@\sigma = s_v@(\sigma \ominus 1)) \vee$$

$$(\bigvee_{i \in I} s_i@\sigma.ts = \sigma \;\wedge\; \exists\, v_\sigma (\bigwedge_{i \in I} s_i@\sigma.ts = \sigma \Rightarrow s_i@\sigma.v = v_\sigma \;\wedge\; s_v@\sigma = (\sigma, v_\sigma)))$$

$$\wedge\, [\![\psi]\!]^u_\delta \;\wedge\; all\_stable(w - s_v, \delta, u)$$

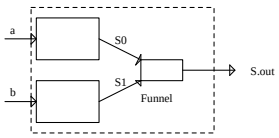

Figure 5: Funnel Example

E.3.4 Equational Theory

1. $s_v \rightsquigarrow s_v \quad = \quad skip$

2. $\{s_v\} : \bigwedge_{\sigma \in [t,t']} (\bigwedge_{i \in I} s_i @ \sigma.ts \neq \sigma \ \wedge \ s_v @ \sigma = s_v @ (\sigma \ominus 1)) \ \vee$

$$(\bigvee_{i \in I} s_i @ \sigma.ts = \sigma \ \wedge \ \exists \, v_\sigma (\bigvee_{i \in I} s_i @ \sigma.ts = \sigma \Rightarrow s_i @ \sigma.v = v_\sigma \ \wedge \ s_v @ \sigma = (\sigma, v_\sigma)))$$

E.4 Case-Study: The Watchdog Timer

In the following case-study we utilise both the current time and funnel agents to provide an efficient concrete agent which implements the watchdog timer (a description of which can be found. in detail in [Hoo91] [Sch90] [Dav91]). We adopt the following specification:

> We assume a finite number of channels d_i with some indexing set I, which are written to by dog processes. Assciated with each d_i is a period p_i and we expect at least one write to occur to each d_i every p_i. If a write fails to occur on one or more d_i then a boolean shunt labelled '*Alarm*' is written within p_j (for the shortest period p_j for which a write failed). Each shunt d_i is monitored for a total of n_i periods.

We can represent the watchdog architecture graphically with the diagram in figure 2.

The properties of the watchdog timer can be defined as a conjunction of properties of individual dog monitors as follows:

$$\begin{aligned} Prop_i \ =_{def} \ & \bigwedge_{\sigma \in [t+p_i, (p_i.n_i)]|p_i} ((stable(d_i, \sigma - p_i, \sigma) \Rightarrow write(alarm_i, true, \sigma, \sigma + p_i)) \wedge \\ & (\neg stable(d_i, \sigma - p_i, \sigma) \Rightarrow stable(alarm, \sigma, \sigma + p_i))) \wedge \\ & stable(alarm_i, p_i.(n_i + 1), t') \end{aligned}$$

Figure 6: Watchdog Timer

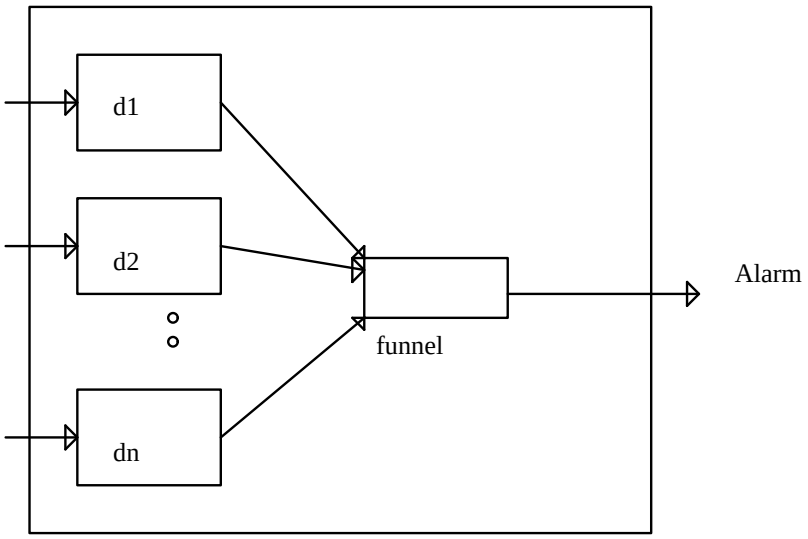

Figure 7: Inner Structure

Along with the property that states that if an individual dog shunt produces an alarm, then the output channel *Alarm* is written to, otherwise it remains stable:

$$Join =_{def} \bigwedge_{\sigma \in [t,t']} ((\bigwedge_{i \in I} alarm_i@\sigma.ts \neq \sigma \Rightarrow Alarm@\sigma = Alarm@(\sigma \ominus 1)) \wedge$$

$$(\bigvee_{i \in I} alarm_i@\sigma = \sigma \Rightarrow \exists\, v_\sigma((\bigwedge_{i \in I} alarm_i@\sigma.ts = \sigma \Rightarrow alarm_i@\sigma.v = v_\sigma) \wedge Alarm@\sigma = (\sigma, v_\sigma)))$$

Clearly the property join suggests a funnel on the output shunts $alarm_i$. The specification for the watchdog timer can now be given by the agent:

$$\{Alarm\} : \exists\, alarm_i(\bigwedge_{i \in I} Prop_i \wedge Join \wedge t' \geq max_i(p_i.(n_i + 1)))$$

The following refinement is a proof sketch only.

We can see that the underlying architecture of the concrete agent which implements this specification will have the structure given in figure 3.

We start by refining each property $Prop_i$ and assert that:

$$\{alarm_i\} : Prop_i \wedge t' \geq max_j(p_j.(n_j + 1))$$

$$\sqsubseteq$$

$$period(wd_i, n_i, p_i, p_i) \mid \delta max_j(p_j.(n_j + 1))$$

where

$$wd_i \;=_{def}\; (x)(t)([1](x \leftarrow d_i \;\mid\; t\downarrow) : t - x.ts > p_i \Rightarrow true \rightarrow alarm_i)$$

and assuming that $\Delta(t - x.ts > p_i) + 2 \leq p_i$. The agent operator 'period' is defined in detail in [SZ92].

This concrete agent is periodic with a period=p_i. At the start of each period it reads the current time and the value found in the dog shunt. The two values are compared and if the dog shunt does not have a fresh enough time-stamp then the alarm is written before the end of the period. The task is therefore checking each previous period for a shunt write.

The property *Join* provides us with the funnel:

$$\{Alarm\} : Join \;\sqsubseteq\; alarm_i \rightsquigarrow_I Alarm$$

Leaving us with the concrete watchdog agent:

$$\sqsubseteq\; (\prod_{i\in I} period(wd_i, n_i, p_i, p_i)) \;\mid\; \delta max_j(p_j.(n_j + 1)) \;\mid\; alarm_i \rightsquigarrow_I Alarm)\; /\{alarm_i\}$$