A Specification Oriented Semantics for the Refinement of Real-T ime Systems

David Scholefield, Hussein Zedan, He Jifeng †

Formal Systems Research Group
Department of Computer Science
University of York
Heslington
York

†Programming Research Group
Oxford University
Keble Road
Oxford

February 16, 1993

Abstract

A refinement calculus for the development of real-time systems is presented. The calculus is based upon a wide-spectrum language called TAM (the Temporal Agent Model), within which both functional and timing properties can be expressed in either abstract or concrete terms. A specification oriented semantics for the language is given. Program development is considered as a refinement process i.e. the *calculation* of a structured program from an unstructured specification. A calculus of decomposition is defined. An example program is developed.

1 Introduction

The formal development of a computer system is traditionally separated into four tasks; the formulation of a specification , the formulation of a design, verification between specification and design, and the translation of the design into an implementation. In real-time systems verification must take into account both functional correctness and the timeliness of results – a consideration which adds further complexity to an already difficult task.

In most formal development methods there are at least two languages involved, one for the specification task, and one for the design task (often the translation to implementation is ignored, or considered to be trivial). However , an inherent problem with such a 'multi language' approach 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 formulate a design, and then verification – a time consuming task – is undertaken. If the verification fails then the design task is undertaken again. This cycle is undergone repeatedly until verification is achieved.

To overcome this problem we have developed the 'Temporal Agent Model' (TAM) which is a theory centered around a wide-spectrum language in which both spec-

ifications and executable programs can be formulated. A real-time functional specification in TAM is transformed step-by-step into a mixed program containing both specification fragments and executable code. Such transformations continue until a completely executable program is produced which is guaranteed correct with respect to the original specification. The program may then be analysed by run-time schedulability and allocation tools in the usual manner , and executed.

The paper introduces a computational model for real-time systems, conservative extensions to first-order predicate logic to cover time, a wide spectrum language with a specificational semantics, a refinement calculus, and an example of program development.

2 A Computational Model for TAM

A computational model should reflect the intended target application area, and also define what we mean by terms such as 'termination', 'communication', 'concurrency', 'functionality', and in our case 'refinement' as well. In doing so, it also defines the class of systems which we may represent and reason about. We provide an informal description of the computational model and discuss the intended target application area.

Most real-time systems can be viewed as a set of periodic and sporadic tasks (assuming some limitations on sporadic inter -arrival rates etc.) [5] [1], and such systems should be easy to represent in the model. In particular the model should enable the developer to describe systems as a set of computational agents with a number of attributes including: release time and period (for periodics), release event and minimal inter -arrival time (for sporadics), functional computation, deadlines on communications and computations, an input and output interface, and, upper -bounds on resource requirements (state-space).

We define a real-time system as a collection of possibly concurrently executing computation agents which communicate asynchronously via time-stamped shared data areas called shunts. Systems can themselves be viewed as single agents and composed into larger systems. Systems 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. the shunt connection topology remains fixed throughout the lifetime of the system. Every system also has the constraint that there can only be a finite number of shunts and agents.

At any instant in time a system can be thought of as having a unique state, which is defined by the values in the shunts and in the local variables of agents; *computation* is therefore defined as any process which results in a change of system state, or ensures explicit state stability . A corollary of these two definitions is that no state change may be instantaneous (take zero time) – a restriction which would seem intuitive.

Time is global i.e. a single clock is accessible to every agent and shunt. The time domain is discrete, 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.

An agent is described by a set of computations which may transform a local data-space and may read and write shunts during execution. The computation may be nondeterministic, and have both minimum and maximum execution times imposed, in particular:

- An agent performs computation and communication; only an agent which performs no computation or communication may be instantaneous.
- 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 computations and communication. Deadlines may be dynamic i.e. dependent upon results of computation (alternate computations may have different deadlines), or static. Deadlines are all considered to be *hard* , that is, there is no concept of deadline priority – all deadlines must be met by the run-time system.
- A data-space is created when an agent starts execution with nondeterministic initial values; the data space is destroyed when the agent terminates. No agent may read or write another agent' s local data-space.
- 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).

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 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 finite 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 assumed 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 any value. When an agent performs a read, the shunt returns both the time-stamp and the associated value. Again, any delay is due to communication overheads.

Shunt readership may be restricted to a set of agents, this set can then be considered as a subsystem where shunts which are read or written by the agents outside 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 a second subsystem which does not wholly contain the former .

The need for time-stamps 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 two different writes. Non-destructive reading 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 e.g. [13].

The local state space of an agent is partitioned and named by computational variables. Each variable is of a pre-determined type (set). We are not especially interested in the mechanics of functional computation in TAM, and we assume that there is some language for functional expression evaluation. 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.

3 TAM Language Syntax

The modular abstraction mechanism in TAM is the *agent* whose syntax can be defined recursively by:

$$\mathcal{A} ::= x := e \mid x \leftarrow s \mid x \rightarrow s \mid w : \Phi \mid \perp\!\!\!\perp \mid (x : T)\mathcal{A} \mid \mathcal{A}|\mathcal{A}$$

$$\mid [S]\mathcal{A} \mid \mathcal{A};\mathcal{A} \mid \bigsqcup_{i \in I}^{t} g_i \Rightarrow \mathcal{A}_i \mid \mathcal{A} \triangleright_t^s \mathcal{A} \mid \mu x_n^S \mathcal{A}$$

Where x is a computation variable, s is a shunt, T and T' are type names, t is a time (constant), I is some indexing set, g_i are boolean expressions, e is an expression on computation variables and shunts, and S is a set of times.

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 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 values of the variables and shunts found in e are taken at the release time of the assignment.

$x \leftarrow s$ performs an input from the shunt s into the variable x. The type of x must therefore be a natural number (time-stamp) and value pair . The input is asynchronous and therefore requires no prior write to the shunt.

$x \rightarrow s$ 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 and is expected to be provided by the run-time environment.

$w : \Phi$ is an agent which specifies required behaviour on variables and shunts. The property Φ is *framed* by the set of variable and shunt names w (which also contains type information). Those variables and shunts which appear in w may be changed during the execution of the agent specified by Φ; those variables and shunts which are not in the frame, but which are inherited from the agent' s

context, may not be changed. There can be two unique free variables of type $Time$ in Φ: t denotes the start of execution and t' denotes the termination time.

$\mathbb{1}$ is an agent which may terminate at any arbitrary time but does not change any variable or shunt.

$(x : T)\mathcal{A}$ introduces the unique variable or shunt x and associates it with the agent $\mathcal{A}$. Initially x has an undefined value (any value from the appropriate type set T). The variable or shunt is destroyed (unavailable) after the termination of $\mathcal{A}$.

$\mathcal{A}|\mathcal{A}$ 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).

$[S]\mathcal{A}$ is an agent which has a duration (a duration of an agent is the interval over which the agent may be active, i.e. the difference between the release time and the termination time) equal to one of the values in the set S. This agent form therefore describes a *deadline* which we expect to be respected by the run-time execution environment.

$\mathcal{A};\mathcal{A}$ defines the sequential composition of the two agents. Immediately upon termination of the first agent the second agent will be released.

$\bigsqcup_{i \in I}{}_t\, g_i \Rightarrow \mathcal{A}_i$ evaluates all of the guards g_i (which are boolean expressions on local state space), and executes one of the agents corresponding to a true guard. If no guards evaluate to true, then the choice agent terminates correctly . The constant t defines an execution window within which the guards are evaluated; it is assumed that the evaluation will take no more than t time units, and that the run-time execution environment will 'pad-out' the evaluation if necessary to exactly fill the interval. The indexing set I is always finite.

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

$\mu x_n^S \mathcal{A}$ executes the agent $\mathcal{A}$ in sequence n times. Each execution of $\mathcal{A}$ is also given the deadline set S.

We impose the syntactic restriction that no agent may share its local state space with a concurrently executing agent, and only one concurrently executing agent may write to any given shunt. An agent is assumed to have an implicit alphabet of local variables, input shunts, and output shunts.

3.1 Derived Agents

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

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 a minimum of n time units is derived as:

Definition Delay . $\delta n =_{def} [\{k | k \geq n\}]\mathbb{1}$

It is also useful to be able to define a nondeterministic choice between agents. We therefore define the syntax:

Definition Choice. $\mathcal{A} \sqcap \mathcal{B} =_{def} (true \Rightarrow \mathcal{A} \sqcup_0 true \Rightarrow \mathcal{B})$

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 number of iterations n:

Definition Period. $Period(\mathcal{A}, p, d, n) = \mu x_n^{\{p\}}(\delta p | [1..d]\mathcal{A})$

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

We also define a shorthand for a agent with a specific termination time.

Definition Termination time. $[n]\mathcal{A} =_{def} [\{n\}]\mathcal{A}$

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 delay agent.

3.2 The Specification Statement

The syntax of the specification agent in TAM is defined as:

$w : \Phi$

where w is a set of *typed* variables and shunts which may be changed during the behaviour defined by the specification, and Φ is a timed logic formula which describes that behaviour . The timed logic formula is a first-order logic formula with simple extensions to cover time. We add terms of the form $x@t$ to denote the value of the variable (or shunt) x at time t. We will also use the notation $s@t.ts$ to refer to the time stamp found in shunt s at time t, and similarly , $s@t.v$ for the value.

In a specification $w : \Phi$, the formula Φ may contain two unique free variables $t, t' \in Time$ which represent the time at which the behaviour described by Φ starts (t), and the time at which the behaviour terminates (t'). Computation variables that appear in Φ may only be referred to at times t and t'; shunts that appear in Φ may be referred to at any times in $[t, t']$. The restriction on variable times supports the intuition that we cannot observe the value of any variables local to an agent during execution, only at the initial and final times.

4 Semantics

We introduce some new notation to simplify the semantic definitions, including a temporal feasibility test (to determine the minimum amount of time an agent needs in which to ensure correct termination), a chop operator , a deadline operator , and correctness axioms for the shunts.

4.1 Pre-defined Predicates

We may also define TAM specific operators which will occur often in specification statements. The predicate '$stable$' asserts that the shunt s will not be changed during the given interval:

Definition Stable (shunts).

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

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.

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

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

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,

$$\mathcal{A}^\frown\mathcal{B} =_{def} \exists m \in [t, t'](\mathcal{A}[m/t'] \wedge \mathcal{B}[m/t])$$

The chop operator has a number of pleasant properties:

$$(Chop.1)\ (\mathcal{A}^\frown\mathcal{B})^\frown\mathcal{C} \equiv \mathcal{A}^\frown(\mathcal{B}^\frown\mathcal{C}) \qquad (Chop.2)\ \mathcal{A}^\frown(\mathcal{B} \vee \mathcal{C}) \equiv (\mathcal{A}^\frown\mathcal{B}) \vee (\mathcal{A}^\frown\mathcal{C})$$
$$(Chop.3)\ (\mathcal{A} \vee \mathcal{B})^\frown\mathcal{C} \equiv (\mathcal{A}^\frown\mathcal{B}) \vee (\mathcal{A}^\frown\mathcal{C}) \qquad (Chop.4)\ \mathcal{I}^\frown\mathcal{A} \equiv \mathcal{A}^\frown\mathcal{I}$$

where $\mathcal{I} =_{def} \emptyset : (t' = t)$

We also define an operator which constrains the value of the free variable t' (the termination time of a property) to a given set:

Definition Deadline.

Given a set of times $S \subseteq Time$, and a timed logic formula $\mathcal{A}$:

$$\mathcal{A}_S =_{def} \mathcal{A} \wedge (t' - t \in S)$$

The deadline operator has the following properties:

$$(Deadline.1)\ (\mathcal{A}_T)_S \equiv \mathcal{A}_{T\cap S} \qquad (Deadline.2)\ (\mathcal{A} \vee \mathcal{B})_T \equiv (\mathcal{A}_T) \vee (\mathcal{B}_T)$$
$$(Deadline.3)\ \mathcal{A}_{Time} \equiv \mathcal{A}$$

4.2 Shunt Axioms

We axiomatize the constraints on the time stamps within the shunts, imposed by the computational model, by the following axioms (which may later be used by the refinement process).

We define a notation which asserts that a consistent time stamp has been written before a given time by:

$$\vec{(}s)_t =_{def} \exists t' \in Time(t' \leq t \ \wedge \ s@t'.ts = t')$$

We may now assert that after the first write to a shunt, all time stamps must have a value before or at the current time. This is true only after the first write because the shunt has a non-determined initial value:

SA.1 $\forall t \in Time(\vec{(}s)_t \Rightarrow s@t.ts \leq t)$

We also assert that if a shunt value changes then the time stamp must change to the current time:

SA.2 $\forall t \in Time(t > 0 \wedge s@t.v \neq s@(t-1).v \Rightarrow s@t.ts = t)$

Finally we assert that a shunt value can only change when the time stamp is equal to the current time:

SA.3 $\forall t \in Time(t > 0 \wedge s@t.ts \neq t \Rightarrow s@t = s@(t-1))$

4.3 Agent Semantics

The semantics of an agent are given by a timed logic formula. The specification statement is defined in this manner also, giving a natural interpretation for a refinement relation.

The semantics for the specification statement assumes that the behaviour specified by Φ removes any obligations in the postcondition based on the assumption that a concrete agent which satisfies the specification is executed. The Σ-feasibility number is checked against the available execution window $[\delta, u]$. The definitions of other agents are then given by a single specification agent, or simple timed logic formula.

We also 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 (i.e. if W is the alphabet of the specification statement $w : \Phi$, then $\tilde{w} =_{def} W/w$). Where variables appear dashed (x') we assume $x@t'$. We also remove the usual semantic parenthesis without loss of clarity .

Definition Semantics.

$w : \Phi \quad =_{def} \quad stable(\tilde{w}, t, t') \ \wedge \ \Phi$

$\perp\!\!\!\perp \quad =_{def} \quad \emptyset : t \leq t'$

$x := e \quad =_{def} \quad \{x : T\} : (t < t' \wedge x' = e)$
(where $type(e) = T$, and all variables in e are at time t)

$x \rightarrow s \quad =_{def} \quad \{s : [Time \times T]\} : write(x, s, t, t')$
(where $type(x) = T$)

$x \leftarrow s \quad =_{def} \quad \{x : [Time \times T]\} : t < t' \wedge \exists m \in (t, t'](x' = s@m)$
(where $type(s) = [Time \times T]$)

$(x : T)\mathcal{A} \quad =_{def} \quad \exists x \in T(\mathcal{A})$

$\mathcal{A}|\mathcal{B} \quad =_{def} \quad (\mathcal{A}^\frown\!\!\perp\!\!\!\perp) \wedge (\mathcal{B}^\frown\!\!\perp\!\!\!\perp)$

$[S]\mathcal{A} \quad =_{def} \quad \mathcal{A}_S$

$\mathcal{A}; \mathcal{B} \quad =_{def} \quad \mathcal{A}^\frown\mathcal{B}$

$\bigsqcup_{i \in I}^{n} g_i \Rightarrow \mathcal{A}_i \ =_{def} \ [n]\perp\!\!\!\perp^\frown(((\bigwedge_{i \in I} \neg g_i) \wedge \perp\!\!\!\perp)) \ \vee \ \bigvee_{i \in I}(g_i \wedge \mathcal{A}_i))$

$\mathcal{A} \triangleright_n^s \mathcal{B} \quad =_{def} \quad Nowrite^\frown\mathcal{A} \ \vee \ Input^\frown B$

$\mu x_{n+1}^S \mathcal{A} \quad =_{def} \quad ([S]\mathcal{A}); \mu_n^S \mathcal{A}$

$\mu x_0^S \mathcal{A} \quad =_{def} \quad \perp\!\!\!\perp$

where:

$Nowrite \ =_{def} \ \emptyset : t' = t + d \ \wedge \ s@t.1 < t \ \wedge \ stable(s, t, t')$

$Input \ =_{def} \ \emptyset : t' \in [t, t + d] \ \wedge \ s@t'.1 = t' \ \wedge \ \forall m \in [t, t')(s@m = s@t)$

5 Refinement

A refinement relation is defined as a *strengthening* of a timed formula. This models the intuitive definition of refinement as a lessening of nondeterminism.

Definition Refinement.

$\mathcal{A} \sqsubseteq \mathcal{B} \quad iff \quad \mathcal{B} \Rightarrow \mathcal{A}$

This definition gives rise to a complete lattice with the (theoretical) TAM agents:

Definition Top and Bottom $\quad \perp \ =_{def} \ true \quad \top \ =_{def} \ false$

defining the top bottom and top of the lattice respectively . Refinement is a path upwards through the lattice, and in particular , the top element is called *miracle* and is a refinement of any program, i.e.

Theorem $\mathcal{A} \sqsubseteq \top$

If the miraculous agent is reached during refinement then is is likely that an inconsistency exists (either temporal or functional) in the original specification.

In addition to the above theorem, a number of sound refinement laws are presented in the appendix.

6 Composition and Decomposition

Composite agents allow us to describe complex behaviour by combining simpler agents. However , in system development we would not want to complicate the refinement method in proportion to the complexity of the desired system, instead we would wish to have simple refinement obligations which could be trivially composed in order to discharge complex refinement proof obligations.

In [8] the importance of compositionality of proof systems for concurrent real-time formalisms is discussed. We assert that the refinement calculus for TAM is compositional i.e. systems can be sub-divided into subsystems which may then be refined in isolation, and recomposed to give a system which is a refinement of the original specification.

This form of compositionality is clearly dependent upon the fact that refinement can not introduce non-local variables and therefore break the interference constraint on concurrent systems, and similarly can not introduce unrestricted shunts.

The property of compositionality also holds for the other agent constructors (deadline, sequence, variable declaration, shunt restriction, timeout, guards and recursion), and in the refinement calculus compositionality equates to the property of *monotonicity* .

We therefore assert the following theorem:

Theorem (Refinement Monotonicity) [11]

> If $\mathcal{A} \sqsubseteq \mathcal{B}$ then for any context $\mathcal{C}(_)$ we have $\mathcal{C}(\mathcal{A}) \sqsubseteq \mathcal{C}(\mathcal{B})$

Although compositionality is important in reducing proof complexity , the process of refinement is one of decomposition, not composition. It is therefore fruitful to investigate the conditions under which agents can be decomposed, i.e. how structured programs can be *calculated* from unstructured specifications.

The problem of decomposition can be described as

> Let R be a given specification, then if one decides to choose $op(X, Y)$ as the architecture of the implementation, the refinement calculus has to find a specification for X and Y.

The solution lies in the definition of a predicate transformer which defines the inverse of an operation. The first example is that of the sequence.

Let P and R be agents (predicates) with identical alphabets, then we define:

Definition Weakest Successor $\quad R \sqsubseteq (P \frown X) \;\; iff \;\; succ(P, R) \sqsubseteq X$

Therefore $succ(P, R)$ is the weakest specification of the successor of P such that their composition refines R.

The natural complement to the weakest successor is the weakest predecessor .

Definition Weakest Predecessor $\quad R \sqsubseteq (X \frown Q) \;\; iff \;\; pred(Q, R) \sqsubseteq X$

We can see that $pred(Q, R)$ is the weakest specification of the predecessor of Q such that their sequential composition meets the specification R.

Finally , the weakest environment operator defines the inverse of the concurrent operator .

Definition Weakest Environment $\quad R \sqsubseteq (P \wedge X) \;\; iff \;\; env(P, R) \sqsubseteq X$

where the alphabets of P and X form the alphabet of R.

Some properties of the inverse operators are given in the appendix.

7 Example

We refine a specification for a simple comparator which reads data from two input shunts, and if the data values are equal, outputs the value 'true' on an output shunt, otherwise it outputs the value 'false'. The comparator must take no longer than 10 time units to compare and output.

In addition we are provided with an axiom which asserts that the value of the input shunts will remain stable during the comparator period:

$A.1 \quad =_{def} \;\; stable(\{in_1, in_2\}, t, t')$

Given three shunts:

$in_1, in_2 \;:\; [Time \times T]$ (input shunts)
$out \;:\; [Time \times bool]$ (output shunt)

then the comparator can be specified by:

$$Spec \;\; =_{def} \;\; \{out : [Time \times bool]\} : t' \leq t + 10 \wedge (\bigwedge_{\sigma \in [t,t']} in_1@\sigma.v = in_2@\sigma.v) \Rightarrow write(out, \text{`}true\text{'}, t, t') \wedge (\neg \bigwedge_{\sigma \in [t,t']} in_1@\sigma.v \neq in_2@\sigma.v) \Rightarrow write(out, \text{`}false\text{'}, t, t')$$

Rule SR.4 allows us to strengthen the specification, and we do so by constraining the termination time further , and appealing to the axiom A.1 to allow us to choose an arbitrary time at which to test the input shunts.

$$Spec \sqsubseteq \{out : [Time \times bool]\} : t' = t + 10 \wedge (in_1@(t+1).v = in_2@(t+1).v) \Rightarrow write(out, `true', t, t') \wedge (in_1@(t+1).v \neq in_2@(t+1).v) \Rightarrow write(out, `false', t, t')$$

The rule SR.4 also allows us to introduce existentially quantified variables which will be used to hold the values read from the shunts.

$$Spec \sqsubseteq \{out : [Time \times bool]\} : t' = t + 10 \wedge \exists r_1, r_2 : [Time \times T](r_1@(t+1) = in_1@(t+1) \wedge r_2@(t+1) = in_2@(t+1) \wedge (r_1@(t+1).v = r_2@(t+1).v) \Rightarrow write(out, `true', t, t') \wedge (r_1@(t+1).v \neq r_2@(t+1).v) \Rightarrow write(out, `false', t, t'))$$

We can move the existential quantification out of the specification and declare them as local variables by VR.4. In addition we can introduce an intermediate time value m (we are clearly moving towards sequential composition).

$$Spec \sqsubseteq (r_1, r_2 : [Time \times T])\{r_1, r_2 : [Time \times T],\ out : [Time \times bool]\} : \exists m \in [t, t'](t' = m + 9 \ \wedge \ m = t + 1 \wedge r_1@m = in_1@m \wedge r_2@m = in_2@m \wedge (r_1@m.v = r_2@m.v) \Rightarrow write(out, `true', t, t') \wedge (r_1@m.v \neq r_2@m.v) \Rightarrow write(out, `false', t, t'))$$

The next refinement allows us to reduce the period during which the write may take place. This is sound by SR.4

$$Spec \sqsubseteq (r_1, r_2 : [Time \times T])\{r_1, r_2 : [Time \times T],\ out : [Time \times bool]\} : \exists m \in [t, t'](t' = m + 9 \ \wedge \ m = t + 1 \wedge r_1@m = in_1@m \wedge r_2@m = in_2@m \wedge stable(out, t, m) \wedge (r_1@m.v = r_2@m.v) \Rightarrow write(out, `true', m, t') \wedge (r_1@m.v \neq r_2@m.v) \Rightarrow write(out, `false', m, t'))$$

We can now refine to a sequential agent (by def ;).

$$Spec \sqsubseteq (r_1, r_2 : [Time \times T])\mathcal{A}; \mathcal{B}$$

where

$$\mathcal{A} =_{def} \{r_1, r_2 : [Time \times T],\ out : [Time \times bool]\} : stable(out, t, t') \ \wedge \ t' = t + 1 \ \wedge \ r_1@t' = in_1@t' \ \wedge \ r_2@t' = in_2@t'$$

$$\begin{array}{lll}\mathcal{B} & =_{def} & \{r_1, r_2 : [Time \times T],\ out : [Time \times bool]\} : \\ & & (r_1@t.v = r_2@t.v) \Rightarrow write(out, `true', t, t') \wedge \\ & & (r_1@t.v \neq r_2@t.v) \Rightarrow write(out, `false', t, t') \wedge \\ & & t' = t + 9\end{array}$$

We refine these two agents independently . Firstly , the shunt out can be removed from $\mathcal{A}$ by SR.3.

$$\mathcal{A} \sqsubseteq \{r_1, r_2 : [Time \times T]\} : t' = t + 1 \ \wedge \ r_1@t' = in_1@t' \ \wedge \ r_2@t' = in_2@t'$$

The agent $\mathcal{A}$ is eventually refined down to a concurrent agent which reads the two shunts (the proof of this refinement is given in the appendix).

$$\mathcal{A} \sqsubseteq [1](r_1 \leftarrow in_1 \mid r_2 \leftarrow in_2)$$

The agent $\mathcal{B}$ is now refined by removing the variables from the frame (appealing to SR.4 by adding a conjunct asserting their stability).

$$\begin{array}{lll}\mathcal{B} & \sqsubseteq & \{out : [Time \times bool]\} : (r_1@t.v = r_2@t.v) \Rightarrow write(out, `true', t, t') \wedge \\ & & (r_1@t.v \neq r_2@t.v) \Rightarrow write(out, `false', t, t') \wedge \\ & & t' = t + 9\end{array}$$

We can reduce the writing interval still further (which will allow us to refine to a guarded agent) by appealing to SR.4

$$\begin{array}{lll}\mathcal{B} & \sqsubseteq & \{out : [Time \times bool]\} : (r_1@t.v = r_2@t.v) \Rightarrow write(out, `true', t + 2, t') \wedge \\ & & (r_1@t.v \neq r_2@t.v) \Rightarrow write(out, `false', t + 2, t') \wedge \\ & & t' = t + 9 \ \wedge \ stable(out, t, t + 2)\end{array}$$

The constraint on the termination time can be replaced by a deadline (by def $[S]\mathcal{A}$).

$$\begin{array}{lll}\mathcal{B} & \sqsubseteq & [9]\{out : [Time \times bool]\} : (r_1@t.v = r_2@t.v) \Rightarrow write(out, `true', t + 2, t') \wedge \\ & & (r_1@t.v \neq r_2@t.v) \Rightarrow write(out, `false', t + 2, t') \wedge \\ & & stable(out, t, t + 2)\end{array}$$

We may now refine to a guard (by def guards).

$$\begin{array}{lll}\mathcal{B} & \sqsubseteq & [9]\ (r_1.v = r_2.v) \Rightarrow \{out : [Time \times bool]\} : write(out, `true', t, t') \\ & & \sqcup_2 \\ & & (r_1.v \neq r_2.v) \Rightarrow \{out : [Time \times bool]\} : write(out, `false', t, t')\end{array}$$

Both guarded agents are then refined by writes.

$$\begin{array}{lll}\mathcal{B} & \sqsubseteq & [9]\ (r_1.v = r_2.v) \Rightarrow `true' \rightarrow out \\ & & \sqcup_2 \\ & & (r_1.v \neq r_2.v) \Rightarrow `false' \rightarrow out\end{array}$$

Placing the two agents back in sequence gives us the final program:

$$\begin{aligned} Spec \quad \sqsubseteq \quad & (r_1, r_2 : [Time \times T])\ [1](r_1 \leftarrow in_1 \mid r_2 \leftarrow in_2)\ ; \\ & [9](\ r_1 = r_2 \Rightarrow \text{`}true\text{'} \rightarrow out \\ & \quad \sqcup_2 \\ & \quad r_1 \neq r_2 \Rightarrow \text{`}false\text{'} \rightarrow out\ ) \end{aligned}$$

8 Conclusions

TAM is unique in providing a wide-spectrum development language for real-time systems in which abstract specifications can be refined down to concrete executable programs. Wide-spectrum languages for non real-time systems have been studied extensively , for example in the SETL language [12], and the CIP project [6], wide-spectrum languages based upon predicate logic are given transformation rules which allow refinement in a manner similar to TAM.

The utility of a wide-spectrum language can be clearly seen in the refinement method used by Morgan in his calculus [10] [9]. In this language, the concrete syntax is provided by an extended version of Dijkstra' s Guarded Command Language (GCL) [7]. The abstract specification syntax is provided by a statement form:

$$w : [pre, post]$$

where 'w' (called the 'frame') defines the scope of the specification, i.e. those state variables which may be changed by the behaviour defined by the specification , and 'pre' and '$post$' are first-order predicate logic formulae which describe the relationship between the program state before the 'execution' of the specification statement, and after the termination of the specification statement respectively . The specification statement can therefore be viewed as a description of the minimum requirements on the behaviour of any concrete statement which may replace it during refinement.

Similarly , in Back and Wright' s wide-spectrum language [4], the concrete code is a version of Dijkstra' s GCL and the specification statement (called an *assert statement*) is denoted $\{b\}$, where b is a formula on the local state. The assert statement will terminate correctly if the local state satisfies the formulae when 'executed', and will abort otherwise.

Original work by Back [Bac78] [2], and later [3], can be seen as the first investigation of adding specification statements to programming languages to aid in the process of verification.

The common factor of both Morgan and Back and Wright' s languages is that they are transformational; they describe computations which have all input data available at the start of execution, and provide the result at the time of termination. This restriction provides the basis for the 'shape' of Morgan' s specification statement – it describes a relationship between initial and final states. In real-time systems we are interested in *reaction* , i.e. input and output during the execution of an agent. In addition, we are interested in the time at which the inputs and outputs occur; our specification statement for real-time systems reflects these requirements.

By providing definitions of inverse operators to assist in the process of decomposition, we are also facilitating the reusability of existing designs. A library of modules could be used in equations which calculated the agent needed to meet a specification. In addition, inverse operators will also enable us to find a complete proof system for TAM.

9 References

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

[2] R. J. R. Back, **"Correctness Preserving Program Refinements: Proof Theory and Applications"** , Tract 131, Mathematisch Centrum, Amsterdam. 1980.

[3] R. J. R. Back, **"A Calculus Of Refinements for Program Derivations"** , Acta-Informatica, 25, p593-624. 1988.

[4] R. J. R. Back, J Wright, **"Refinement Concepts Formalised In Higher Order Logic"** , BCS-F ACS, Vol 2, No.3. 1990.

[5] A. Burns, A. Wellings, **"Real-T ime Systems And Their Programming Languages"** , Addison-W esley . 1990.

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

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

[8] J. Hooman, **"Specification and Compositional Verification of Real-T ime Systems"** , Ph.D. Thesis, Technical University of Eindhoven. 1991.

[9] C. Morgan, **"Programming From Specifications"** . PHI. 1990.

[10] C. Morgan, K. Robinson, P. Gardiner , **"On The Refinement Calculus"** , PRG-70, Oct 1988.

[11] D. Scholefield, **"A Refinement Calculus for Real-T ime Systems"** , Department of Computer Science D.Phil Thesis. University of York. July 1992.

[12] 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.

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

A Refinement Laws and Proof

A.1 Refinement Laws

The following refinement laws are sound [11].

Specification

SR.1 $w : \Phi \;=\; w : \Phi[x@t/x@t']$ (if $x \notin w$)

SR.2 $w : \Phi \;=\; w : \Phi[x@t'/x@t]$ (if $x \notin w$)

SR.3 $w \cup \{s : T\} : \Phi \;\wedge\; stable(s, t, t') \;=\; w - (s : T) : \Phi$

(if s not in Φ)

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

Restriction

For any variable x:

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

$VR.2\ (x : T)((y : T')\mathcal{A}) \;=\; (y : T')((x : T)\mathcal{A})$

$VR.3\ w : (\exists x : T(\Phi)) \;=\; (x : T)(w \cup \{x : T\} : \Phi)$ (if $x \notin w$)

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

(if x is new unique computational variable)

Assignment

AR.1 $x := x \;=\; \delta 1$

Input

IR.1 $(x : [Time \times T])(x \leftarrow s) \;=\; \delta 1$

Output

OR.1 $\{s : T\} : write(s, x@t, t, t') \;=\; x \rightarrow s$

Sequential

SE.1 $P; \mathcal{I} \;=\; \mathcal{I}; P \;=\; P$

SE.2 $P; (Q; R) \;=\; (P; Q); R$

SE.3 $P; (Q \sqcap R); S \;=\; (P; Q; S) \sqcap (P; R; S)$

Deadline

DE.1 $[S_1 \cup S_2]\mathcal{A} \;=\; ([S_1]\mathcal{A} \sqcap [S_2]\mathcal{A})$

Concurrent

CR.1 $\mathcal{A}|\mathcal{B} \;=\; \mathcal{B}|\mathcal{A}$

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

CR.3 $\mathcal{A}|(\mathcal{B} \sqcap \mathcal{C}) \;=\; (\mathcal{A}|\mathcal{B}) \sqcap (\mathcal{A}|\mathcal{C})$

Guards

GR.1 $b_1 \Rightarrow \mathcal{A} \sqcup_t b_2 \Rightarrow \mathcal{B} = b_2 \Rightarrow \mathcal{B} \sqcup_t b_1 \Rightarrow \mathcal{A}$

GR.2 $true_t \Rightarrow \mathcal{A} = [\{t\}]\delta t; \mathcal{A}$

GR.3 $false_t \Rightarrow \mathcal{A} = \delta t$

GR.4 $b_1 \Rightarrow \mathcal{A} \sqcup_t b_2 \Rightarrow \mathcal{A} = (b_1 \vee b_2)_t \Rightarrow \mathcal{A}$

GR.5 $[S](\bigsqcup_{i \in I_t} g_i \Rightarrow \mathcal{A}_i); \mathcal{B} = \bigsqcup_{i \in I_t} g_i \Rightarrow [S \ominus t]\mathcal{A}_i; \mathcal{B}$

Where $S \ominus t =_{def} \{s \mid exists s' \in S(s' - t = s \wedge s \geq 0)\}$

$(if\ t \leq max(S)$ and $\bigvee_{i \in I} g_i)$

GR.6 $[S](\bigsqcup_{i \in I_t} g_i \Rightarrow \mathcal{A}_i) : \mathcal{B} = false$

if $t > max(S)$

GR.7 $(x : T)(\bigsqcup_{i \in I_t} g_i \Rightarrow \mathcal{A}_i) = \bigsqcup_{i \in I_t} g_i \Rightarrow (x : T)\mathcal{A}_i$

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

GR.8 $w : \Phi \sqsubseteq \bigsqcup_{i \in I_\sigma} g_i \Rightarrow w : \Phi_i$

if $\Phi \models t + \sigma \leq t' \wedge$

$((\bigwedge_{i \in I} \neg g_i @ t \wedge all_stable(w, t, t')) \vee$

$(\bigvee_{i \in I} g_i @ t \wedge$

$\bigvee_{i \in I} (g_i @ t \wedge \phi_i[\sigma + 1/t] \wedge all_stable(w, t, t + \sigma))))$

Timeout

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

A.2 Comparator Refinement

In the example refinement we assumed that

$\{r_1, r_2 : [Time \times T]\} : t' = t + 1 \wedge r_1 @ t' = in_1 @ t' \wedge r_2 @ t' = in_2 @ t'$

$\sqsubseteq [1](r_1 \leftarrow in_1 \mid r_2 \leftarrow in_2)$

We now complete this refinement proof. The first step is to separate the two input shunt reads. This is achieved by appealing to SR.4

$$\{r_1, r_2 : [Time \times T]\} : \; t' = t + 1 \;\wedge\; r_1@t' = in_1@t' \;\wedge\; r_2@t' = in_2@t'$$

$$\sqsubseteq \quad \{r_1 : [Time \times T\} : (r_1@t' = in_1@t') \wedge \{r_2 : [Time \times T]\} : (r_2@t' = in_2@t') \wedge t' = t+1$$

Next, intermediate time variables can be introduced to allow for the sequential composition within the concurrency definition. Again we use SR.4

$$\begin{array}{ll} \sqsubseteq & \exists m_1 \in [t,t'](\{r_1 : [Time \times t]\} : \bigvee_{\sigma \in (t,m_1]} (r_1@m_1 = in_1@m_1 \;\wedge\; r_1@t' = r_1@m_1) \wedge \\ & \exists m_2 \in [t,t'](\{r_2 : [Time \times t]\} : \bigvee_{\sigma \in (t,m_2]} (r_2@m_2 = in_2@m_2 \;\wedge\; r_2@t' = r_2@m_2) \wedge \\ & t' = t + 1 \end{array}$$

The timing constraint can be moved out into a deadline.

$$\begin{array}{ll} \sqsubseteq & [1]\,(\exists m_1 \in [t,t'](\{r_1 : [Time \times t]\} : \bigvee_{\sigma \in (t,m_1]} (r_1@m_1 = in_1@m_1 \;\wedge\; r_1@t' = r_1@m_1) \wedge \\ & \quad \exists m_2 \in [t,t'](\{r_2 : [Time \times t]\} : \bigvee_{\sigma \in (t,m_2]} (r_2@m_2 = in_2@m_2 \;\wedge\; r_2@t' = r_2@m_2)) \end{array}$$

The intermediate times introduce $\mathbb{1}$ agents.

$$\begin{array}{ll} \sqsubseteq & [1]\,(\exists m_1 \in [t,t'](\{r_1 : [Time \times t]\} : \bigvee_{\sigma \in (t,m_1]} (r_1@m_1 = in_1@m_1 \;\wedge\; \mathbb{1}) \wedge \\ & \quad \exists m_2 \in [t,t'](\{r_2 : [Time \times t]\} : \bigvee_{\sigma \in (t,m_2]} (r_2@m_2 = in_2@m_2 \;\wedge\; \mathbb{1})) \end{array}$$

The disjunctions give rise to inputs (by def $\leftarrow$).

$$\begin{array}{ll} \sqsubseteq & [1]\,(\exists m_1 \in [t,t'](r_1 \leftarrow in_1 \;\wedge\; \mathbb{1}) \wedge \\ & \quad \exists m_2 \in [t,t'](r_2 \leftarrow in_2 \;\wedge\; \mathbb{1})) \end{array}$$

The inner conjunctions give rise to chops (def $\frown$).

$$\begin{array}{ll} \sqsubseteq & [1]\,((r_1 \leftarrow in_1 \frown \mathbb{1}) \wedge \\ & \quad (r_2 \leftarrow in_2 \frown \mathbb{1})) \end{array}$$

Which is a concurrent agent.

$$\sqsubseteq \quad [1]\,(r_1 \leftarrow in_1 \mid r_2 \leftarrow in_2) \quad \Box$$

A.3 Properties of $succ$

L.1 $P \frown succ(P, R) \;\sqsupseteq\; R$

L.2 $succ(\mathcal{I}, R) \;=\; R$

L.3 $succ(P \frown Q) \;=\; succ(Q, succ(P, R))$

L.4 $succ(P \vee Q) \;=\; succ(P, R) \;\wedge\; succ(Q, R)$

L.5 $succ(P, Q \wedge R) \;=\; succ(P, Q) \;\wedge\; succ(P, R)$

L.6 $succ(P \wedge Q) \;\sqsubseteq\; succ(P, R) \;\vee\; succ(Q, R)$

L.7 $succ(P, Q \vee R) \;\sqsubseteq\; succ(P, Q) \;\vee\; succ(P, R)$

A.4 Properties of $pred$

L.1 $pred(Q, R)^\frown Q \sqsupseteq R$

L.2 $pred(\mathcal{I}, R) = R$

L.3 $pred(P^\frown Q, R) = pred(P, pred(Q, R))$

L.4 $pred(P \vee R, R) = pred(Q, R) \wedge pred(P, R)$

L.5 $pred(P, Q \wedge R) = pred(P, Q) \wedge pred(P, R)$

L.6 $pred(P \wedge Q, R) \sqsubseteq pred(Q, R) \vee pred(P, R)$

L.7 $pred(P, Q \vee E) \sqsubseteq pred(P, Q) \vee pred(P, R)$

A.5 Commutation of $succ$ and $pred$

Theorem $succ(P, pred(Q, R)) = pred(Q, succ(P, R))$

Proof

We start by refining the LHS to give:

$\equiv (P^\frown X) \sqsupseteq pred(Q, R)$

$\equiv (P^\frown X)^\frown Q \sqsupseteq R$

$\equiv P^\frown (X^\frown Q) \sqsupseteq R$

$\equiv (X^\frown Q) \sqsupseteq succ(P, R)$

$\equiv X \sqsupseteq RHS \quad \Box$