Refinement of Complex Systems: A Case Study

G. Lowe*   H. Zedan†

July 10, 1995

Abstract

We describe the Temporal Agent Model (TAM) together with its associated refinement calculus. The calculus is based on a wide-spectrum language within which functional and temporal properties can be expressed in either abstract (i.e. specification) or concrete (i.e. design) terms. The refinement process transforms abstract specifications to concrete designs through successive applications of sound refinement laws. An extension to the calculus allows us to calculate a scheduler for the resulting design. We present a specification paradigm based on splitting the functional and temporal requirements, and describe refinement techniques based on this paradigm. We illustrate the calculus with an example taken from the avionics industry.

1 Introduction

The formal development of a computer system is traditionally separated into four distinct tasks: the formalization of the user's informal requirements, the formalization of the design, verification of the design (i.e. proving that the design satisfies the specification), and the translation of the design into an implementation. For real-time systems, both specification and verification stages must take into account the various non-functional requirements—such as timing and dependability—as well as the usual functional requirements. This consideration adds more complexity to an already complex task.

In recent years, efforts have been mounted to construct *practical* formal techniques for the development of real-time systems. In most existing formal development methods there are at least two languages involved: one for writing specifications, and the other for the design. (The transformation from design to actual implementation is often ignored or considered trivial.) A common problem with this multi-language approach is the lack of method by which a suitable design can be found. A combination of experience and guesswork must be used in order to formulate the design before verification can be undertaken. If this verification fails, then the whole design and verification task is repeated.

In this paper we present the Temporal Agent Model (TAM) as an attempt to overcome these problems. TAM is a theory centered around a wide-spectrum language in which both abstract (i.e. non-implementable) specification and design can be formulated in a unified manner. Both functional and timing requirements of a real-time system are expressed as a TAM specification; this is then transformed,

*Programming Research Group, Oxford University Computing Laboratory, Parks Road, Oxford, OX1 3QD; e-mail `gavin.lowe@comlab.ox.ac.uk`

†School of Computing and Mathematical Science, John Moores University, Liverpool; e-mail `H.S.ZEDAN@livjm.ac.uk`

using provably correct transformations, via a mixed program containing both fragments of abstract specification and executable code, before eventually reaching pure code. This type of development is widely known as refinement, and is already in common use in untimed systems.

The advantage of the wide spectrum language is that the development process remains within the framework of a single language, thus avoiding translation errors which are common in a multi-language approach. The refinement calculus, which allows for the gradual transformation from abstract specification to concrete design, gives a verification methodology which is easy to follow in a step by step fashion.

1.1 Related work

For untimed systems, a refinement calculus was developed by Back [Bac88] to provide a formal framework for stepwise refinement of sequential programs. It extends Dijkstra's weakest precondition semantics for total correctness of programs with a relation of refinement between program statements. This relation is defined in terms of the weakest preconditions of statements, and expresses the requirement that a refinement must preserve total correctness of the statement being refined. The refinement calculus was extended to provide a framework for total correctness for parallel systems in [BW90, Bac89]. Morgan's work [Mor90] provides a good overview of how to apply the refinement calculus in practical program derivations.

A number of refinement calculi already exist for real-time systems, but they are either incomplete or use an unrealistic computational model. PL^{time} [He91] is a real-time design language which consists of a CSP-like syntax with extensions for real-time. However, the formalism is based on the maximal-parallelism hypothesis (i.e. the assumption that there are always sufficient resources available) which is too restrictive for most real time systems. In addition, since PL^{time} does not provide a separate specification statement as a syntactic entity, the refinement remains purely in the concrete domain. Similarly, RT-ASLAN [AK86] is a refinement calculus which refines a specification into concrete code, but this again relies on the maximal parallelism model.

Refinement in Z for purely transformational systems is already common [McD89] but recent work by Hayes and Mahony [MH92] has investigated extensions within the Z specification language to include real-time notation and high-level real-time refinement calculus. Functions from a time domain onto a value set models the value of timed variables at specific times. A specification statement form is added to the notation, and is used to describe the behaviour of systems in terms of the assumed and required behaviour of timed variables. The use of Z notation allows Hayes and Mahony to perform a degree of compositionality with the specification statement, but to an extent this compositionality is artificial; each specification describes an aspect of system behaviour over the entire lifetime of the system and there is no temporal compositionality. Refinement is defined as a relation over specification statements and follows Morgan's original calculus [Mor90]. Rules for local variable introduction, concurrency and piping are also provided. All of these operators are Z schema operators, and the refinement process remains at the specification level; this is not a wide-spectrum refinement calculus.

1.2 Paper organization

In the next section we give an overview of the syntax and semantics of the Temporal Agent Model. In section 3 we present a method of separating the functional and temporal parts of a specification, and also present some results that help transform such specifications into code. In section 4 we present our case study: we informally describe our system, present a formal specification, derive a design meeting this specification, and derive a scheduler for the design.

2 The Temporal Agent Model

We define a real-time system to be a finite collection of concurrently executing computation agents which communicate asynchronously via time-stamped shared data areas called shunts. Shunts are passive shared memory spaces that contain two values: the first gives the time at which the most recent write took place, and the second gives the value that was most recently written. Each shunt is owned by a single agent that can write to it; all agents can then read this value.

Systems themselves can be viewed as single agents and composed into larger systems.

We use a discrete, linear time domain, modelled by the natural numbers. This has the considerable advantage of simplifying our calculations, while losing very little in generality.

At any time, a system can be thought of having a unique state, defined by the values in the shunts and local variables. 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.

The TAM language is defined by[1]:

$$\begin{array}{rl} \mathcal{A} ::= & w : \Phi \;\big|\; \mathsf{skip} \;\big|\; \Delta t \;\big|\; x := e \;\big|\; x \leftarrow s \;\big|\; e \rightarrow s \;\big|\; \mathcal{A} \,;\mathcal{A}' \;\big| \\ & \mathsf{var}\, x : T \;\mathsf{in}\; \mathcal{A} \;\big|\; \mathsf{shunt}\, s : T \;\mathsf{in}\; \mathcal{A} \;\big|\; [t]\,\mathcal{A} \;\big|\; \mathsf{if}_t \;\underset{i \in I}{\Box}\; g_i \Rightarrow \mathcal{A}_i \;\;\mathsf{fi} \;\big| \\ & \mathcal{A} \sqcap \mathcal{A}' \;\big|\; \mathcal{A} \rhd_t^s \mathcal{A}' \;\big|\; \mathcal{A} \mid \mathcal{A}' \;\big|\; \mathsf{loop\,for}\; n \;\mathsf{period}\; t \;\; \mathcal{A}. \end{array}$$

where w is a set of computation variables and shunts; Φ is a predicate in the *TAM Logic Language*, which we define below; t is a time, which we take to be a natural number; x is a variable of type T; e is an expression on variables; s is a shunt of type $Time \times T$; I is some finite indexing set; g_i is a boolean expression; and n is a natural number.

Informally:

- $w : \Phi$ is a specification statement. It specifies that only the variables in the *frame* w may be changed, and the execution must satisfy Φ.
- The agent skip may terminate after any delay.
- The agent Δt terminates after t time units.
- $x := e$ evaluates the expression e, using the values found in variables at the start time of the agent, and assigns it to x. The expression e may not include the values held in shunts: it may only use the values held in variables. It may also include the distinguished free variable Now, representing the current time.
- $x \leftarrow s$ performs an input from shunt s, storing the value in x; the type of x must be a time–value pair.
- $e \rightarrow s$ writes the current value of expression e to shunt s, time-stamping it with the time of the write.
- $\mathcal{A} \,; \mathcal{A}'$ performs a sequential composition of $\mathcal{A}$ and $\mathcal{A}'$.

[1] In earlier papers, TAM was given a more abstract syntax, whose conciseness helped in abstract reasoning; in this paper, where we are dealing with more concrete agents, we adopt a more "user-friendly" syntax, similar to that in [Sch93]

- $\mathsf{var}\, x : T \;\mathsf{in}\; \mathcal{A}$ defines x to be a new local variable of type T within $\mathcal{A}$; its initial value is chosen nondeterministically.

- $\mathsf{shunt}\, s : T \;\mathsf{in}\; \mathcal{A}$ defines s to be a new local shunt of type $\mathit{Time} \times T$ within $\mathcal{A}$; its initial value is chosen nondeterministically, but it is time-stamped with the time of its declaration.

- $[t]\, \mathcal{A}$ gives agent $\mathcal{A}$ a duration of t: if the agent terminates before t seconds have elapsed, then the agent should idle to fill this interval; if the agent does not terminate within t seconds, then it is considered to have failed.

- $\mathsf{if}_t \;\; \underset{i \in I}{\Box}\; g_i \Rightarrow \mathcal{A}_i \;\; \mathsf{fi}$ evaluates all the boolean guards g_i, and executes an $\mathcal{A}_i$ corresponding to a true guard; if all the guards evaluate to false, then the agent terminates correctly. The evaluation of the guards should take precisely t time units; if necessary, the agent should idle to fill this interval. We shall sometimes omit the parameter t if we do not want to specify it. We shall sometimes write this construct as $\mathsf{if}_t\; g_1 \Rightarrow \mathcal{A}_1 \;\Box\; g_2 \Rightarrow \mathcal{A}_2 \;\Box\; \ldots \;\Box\; g_n \Rightarrow \mathcal{A}_n \;\mathsf{fi}$.

- $\mathcal{A} \sqcap \mathcal{A}'$ forms a nondeterministic choice between $\mathcal{A}$ and $\mathcal{A}'$.

- $\mathcal{A} \rhd_t^s \mathcal{A}'$ monitors shunt s for t time units: if a write occurs within this time, then it executes $\mathcal{A}'$; otherwise it times-out and executes $\mathcal{A}$.

- $\mathcal{A} \mid \mathcal{A}'$ executes the two agents concurrently, terminating when both agents terminate.

- $\mathsf{loop\,for}\; n \;\mathsf{period}\; t \;\; \mathcal{A}$ executes $\mathcal{A}$ n times, giving each a duration of t.

We insist that no agent may share its local state space with concurrently executing agents, and only one concurrent agent may write to any given shunt: these restrictions will allow us to develop a compositional semantics and refinement calculus.

2.1 Semantics

The semantics of TAM is given in terms of a predicate in the *TAM Logic Language* (TAMLL). This is a first-order logic with simple extensions to deal with times and the values held in variables and shunts. Formulae in the TAM Logic Language may include two distinguished variables: t_α representing the start time of the execution; and t_ω representing the termination time. It may also use terms of the form $x@t$ and $s@t$ to refer to the values held in variables and shunts at time t. We also use $s@t.ts$ and $s@t.v$ to refer to the time-stamp and value of a shunt.

We define a few useful predicates. The predicate $\mathsf{stable}(x, t, t')$ specifies that the shunt or variable x remains constant between t and t':

$$\mathsf{stable}(x, t, t') \;\widehat{=}\; \forall\, t'' \in [t, t'] \bullet x@t'' = x@t.$$

We extend this to sets of variables and shunts in the obvious way.

The predicate $\mathsf{write}(v, s, t)$ specifies that value v is written to shunt s at time t:

$$\mathsf{write}(v, s, t) \;\widehat{=}\; s@t = (t, v).$$

We extend this to specify that a (single) write occurs during the interval $[t, t']$:

$$\begin{aligned}\mathsf{write}(v, s, [t, t']) \;\widehat{=}\; \exists\, t'' \in [t, t'] \bullet\; & \mathsf{stable}(s, t, t'' - 1) \\ & \wedge \mathsf{write}(v, s, t'') \\ & \wedge \mathsf{stable}(s, t'', t').\end{aligned}$$

Shunts obey the following axiom:

$$\begin{aligned}\forall t \in \mathit{Time} \bullet\ & s@t.ts \leqslant t \\ & \wedge\ s@t.v \neq s@(t-1).v \Rightarrow s@t.ts = t \\ & \wedge\ s@t.ts \neq t \Rightarrow s@t = s@(t-1).\end{aligned}$$

The time stamp represents the last time at which a write was made, so must be no greater than the current time; if the value at time t is not the same as at time $t-1$ then a write must have occurred at t, so the shunt is time-stamped at t; if the time stamp at t is not equal to t, then no write occurred at t so the value is the same as at $t-1$.

We define a semantic function:

$$\mathcal{F} \ : \ \mathit{TAM} \rightarrow \mathit{TAMLL},$$

such that $\mathcal{F}[\![\mathcal{A}]\!]$ gives the semantics of $\mathcal{A}$. The complete semantic definitions are given in [SZH94]. We give here a few examples.

The semantics of the specification statement is given by:

$$\mathcal{F}[\![w : \Phi]\!] \ \widehat{=}\ \mathsf{stable}(W \setminus w, t_\alpha, t_\omega) \wedge \Phi.$$

Here and henceforth we take the agent's alphabet to be W. All variables except those in w (i.e. $W \setminus w$) must remain stable, and the execution must satisfy Φ.

Assignment is defined by:

$$\mathcal{F}[\![x := e]\!] \ \widehat{=}\ \mathsf{stable}(W \setminus x, t_\alpha, t_\omega) \wedge x@t_\omega = e@t_\alpha \wedge t_\omega > t_\alpha.$$

All variables other than x remain stable; the final value of x is the value of e evaluated at t_α; and the agent takes some strictly-positive amount of time to perform the assignment.

Let W_1 and W_2 be the alphabets of $\mathcal{A}_1$ and $\mathcal{A}_2$ respectively. Parallel composition is defined by:

$$\begin{aligned}\mathcal{F}[\![\mathcal{A}_1 \mid \mathcal{A}_2]\!] \ \widehat{=}\ \exists\, t_1, t_2 \bullet\ & \mathcal{F}[\![\mathcal{A}_1]\!][t_1/t_\omega] \wedge \mathsf{stable}(W_1, t_1, t_\omega) \\ & \wedge\ \mathcal{F}[\![\mathcal{A}_2]\!][t_2/t_\omega] \wedge \mathsf{stable}(W_2, t_2, t_\omega) \\ & \wedge\ t_\omega = \max\{t_1, t_2\}.\end{aligned}$$

If $\mathcal{A}_1$ terminates at t_1 and $\mathcal{A}_2$ terminates at t_2 then the parallel composition terminates at $\max\{t_1, t_2\}$; each agent's variables are stable after that agent terminates.

We define refinement in TAM as follows:

$$\mathcal{A} \sqsubseteq_{\mathcal{F}} \mathcal{A}' \ \widehat{=}\ \mathcal{F}[\![\mathcal{A}]\!] \Leftarrow \mathcal{F}[\![\mathcal{A}']\!].$$

If $\mathcal{A}'$ is more fully defined than $\mathcal{A}$, then $\mathcal{A}'$ refines $\mathcal{A}$. (We drop the subscript $\mathcal{F}$ where this will not lead to confusion.)

It is easily shown that refinement is a pre-congruence: that is, if $\mathcal{A} \sqsubseteq \mathcal{A}'$ then $C(\mathcal{A}) \sqsubseteq C(\mathcal{A}')$ for any context $C(\_)$ written in terms of the TAM language. This means that if we refine one part of a program, then we refine the whole program.

2.2 Scheduling-oriented model for TAM

In [Low94], we extended the model covering functionality to include scheduling and resource requirements. We represent the use of resource r at time t by the pair (t, r):

$$\mathit{Resource} \ \widehat{=}\ \mathit{Time} \times \mathit{Resource\_Name}.$$

We alter the syntax so that all constructs requiring a scheduling decision are labelled:

$$\begin{array}{rl} \mathcal{A} ::= & w : \Phi \mid \mathsf{skip} \mid \Delta t \mid L :: x := e \mid L :: x \leftarrow s \mid L :: e \rightarrow s \mid \mathcal{A} ; \mathcal{A}' \mid \\ & \mathsf{var}\, x : T \,\mathsf{in}\, \mathcal{A} \mid \mathsf{shunt}\, s : T \,\mathsf{in}\, \mathcal{A} \mid [t]\, \mathcal{A} \mid L :: \mathsf{if}_t \ \underset{i \in I}{\Box}\, g_i \Rightarrow \mathcal{A}_i \ \mathsf{fi} \mid \\ & \mathcal{A} \sqcap \mathcal{A}' \mid L :: \mathcal{A} \rhd^s_t \mathcal{A}' \mid \mathcal{A} \mid \mathcal{A}' \mid \mathsf{loop\,for}\, n\, \mathsf{period}\ t\ \ \mathcal{A}. \end{array}$$

where L ranges over the space Lbl of labels.

We define a scheduler to be a function ϕ that takes a label and the time at which the program reaches that label, and returns the set of resources upon which the corresponding agent should be scheduled:

$$Sch \ \widehat{=}\ Time \times Label \rightarrow \mathbb{P}(Resource).$$

For example, if the labelled assignment $L :: x := 3$ is encountered at time 7, with scheduler ϕ such that $\phi(7, L) = \{(9, r), (11, r)\}$, then the assignment will be scheduled on resource r at times 9 and 11. Designing a scheduler then amounts to defining the function ϕ for each label, i.e. deciding when, and upon which resources, each atomic construct should be scheduled.

We consider a *system* to be either a specification statement or a concrete agent with a scheduler:

$$\mathcal{S} ::= w : \Phi \mid \mathcal{A} \mid \phi.$$

In order to model scheduling and resource requirements, we extended the TAM Logic Language. We allow our predicates to include a distinguished free variable, R_{req}, representing the resources required by the agent. We also add a boolean-valued variable fail which is true precisely when an execution fails: this failure could be because (1) the agent tries to use a resource for two different purposes, or (2) a deadline is not met. We define a semantic function $\mathcal{R}$ giving the representation of an agent in this model:

$$\mathcal{R} : System \rightarrow TAMLL.$$

For example, the specification statement is extended to specify that the system does not fail:

$$\mathcal{R}[\![w : \Phi]\!] \ \widehat{=}\ \mathsf{stable}(W \setminus w, t_\alpha, t_\omega) \wedge \Phi \wedge \neg\, \mathsf{fail}.$$

For most agents, the semantic definition is similar to the definition in the model without scheduling, extended to include the resource requirements. For example:

$$\begin{array}{rl} \mathcal{R}[\![L :: x := e \mid \phi]\!] \ \widehat{=} & \mathsf{stable}(W \setminus x, t_\alpha, t_\omega) \wedge x @ t_\omega = e @ t_\alpha \\ & \wedge\, R_{req} = \phi(t_\alpha, L) \wedge t_\omega = \mathrm{end}\, R_{req} \wedge \neg\, \mathsf{fail}. \end{array}$$

If the labelled assignment is encountered at time t_α, then it requires resources $R_{req} = \phi(t_\alpha, L)$; it terminates when it has used all these resources; it does not fail.

Consider the parallel composition $\mathcal{A}_1 \mid \mathcal{A}_2$. Suppose $\mathcal{A}_1$ requires resources $R_{req,1}$ and $\mathcal{A}_2$ requires resources $R_{req,2}$; then the parallel composition will require $R_{req,1} \cup R_{req,2}$. The composition will fail if (1) either component fails, or (2) both components try to access the same resource simultaneously.

$$\begin{array}{rl} \mathcal{R}[\![\mathcal{A}_1 \mid \mathcal{A}_2 \mid \phi]\!] \ \widehat{=} & \exists\, t_1, t_2 : Time ; \mathsf{fail}_1, \mathsf{fail}_2 : Bool ; R_{req,1}, R_{req,2} : Resources \bullet \\ & \mathcal{R}[\![\mathcal{A}_1 \mid \phi]\!][t_1 / t_\omega ,\ \mathsf{fail}_1 / \mathsf{fail} ,\ R_{req,1} / R_{req}] \\ & \wedge\, \mathcal{R}[\![\mathcal{A}_2 \mid \phi]\!][t_2 / t_\omega ,\ \mathsf{fail}_2 / \mathsf{fail} ,\ R_{req,2} / R_{req}] \\ & \wedge\, \mathsf{stable}(W_1, t_1, t_\omega) \wedge \mathsf{stable}(W_2, t_2, t_\omega) \\ & \wedge\, t_\omega = t_1 \sqcup t_2 \wedge R_{req} = R_{req,1} \cup R_{req,2} \\ & \wedge\, \mathsf{fail} \Leftrightarrow \mathsf{fail}_1 \vee \mathsf{fail}_2 \vee R_{req,1} \cap R_{req,2} \neq \emptyset. \end{array}$$

The semantic definitions satisfy the following condition, namely that agents only use resources during their execution time:

$$\mathcal{R}[\![\mathcal{A} \mid \phi]\!] \Rightarrow \Big((t, r) \in R_{req} \Rightarrow t \in [t_\alpha, t_\omega)\Big).$$

We can use the above model to define another semantic model, which models the fact that there may be limited resources available. We add another free variable to the TAM Logic Language, R_{av}, representing the resources available to the system. We then define a semantic function

$$\mathcal{L} \;:\; \mathit{System} \to \mathit{TAMLL},$$

by

$$\mathcal{L}[\![\mathcal{S}]\!] \;\widehat{=}\; \exists \mathsf{fail}' \bullet \mathcal{R}[\![\mathcal{S}]\!][\mathsf{fail}'/\mathsf{fail}] \wedge (\mathsf{fail} \Leftrightarrow \mathsf{fail}' \vee R_{req} \not\subseteq R_{av}).$$

Thus systems in this model can fail if they require resources that are not available. Refinement in this model is defined in the obvious way:

$$\mathcal{A} \sqsubseteq_{\mathcal{L}} \mathcal{A}' \;\; \mathit{iff} \;\; \mathcal{L}[\![\mathcal{A}]\!] \Leftarrow \mathcal{L}[\![\mathcal{A}']\!].$$

In [Low94] we proved the following theorem.

Theorem 2.1: Suppose we are given a set of available resources R_{av}, and suppose the variables R_{av} and R_{req} do not appear free in the predicate Φ. Then if agent $\mathcal{A}_0$ meets specification Φ in the non-resource model, agent $\mathcal{A}_1$ can be obtained from $\mathcal{A}_0$ by adding labels, and the composition of $\mathcal{A}_1$ with the scheduler does not fail; i.e.:

$$\begin{array}{rcl} w : \Phi & \sqsubseteq_{\mathcal{F}} & \mathcal{A}_0, \\ \mathcal{A}_1 & \mathit{labels} & \mathcal{A}_0, \\ \mathcal{L}[\![\mathcal{A}_1 \mid \phi]\!] & \Rightarrow & \neg\, \mathsf{fail}, \end{array}$$

then the system meets the specification Φ in the resource model:

$$w : \Phi \;\sqsubseteq_{\mathcal{L}}\; \mathcal{A} \mid \phi.$$

This theorem allows us to split the development of a system into two stages:

- In the first stage we use the standard model and refinement rules to derive an initial design, i.e. a program without a scheduler;
- In the second stage we design a scheduler for the system.

The theorem says that if the scheduling is done correctly—i.e. the system never fails—then the complete system derived in the second stage still meets the functional specification as in the first stage.

The following lemma will be very useful in proving the third of the above conditions:

Lemma 2.2: If the agents $\mathcal{A}_1$ and $\mathcal{A}_2$ with scheduler ϕ use resources R_1 and R_2, respectively, and do not fail:

$$\begin{array}{rcl} \mathcal{R}[\![\mathcal{A}_1 \mid \phi]\!] & \Rightarrow & R_{req} \subseteq R_1 \wedge \neg\, \mathsf{fail}, \\ \mathcal{R}[\![\mathcal{A}_2 \mid \phi]\!] & \Rightarrow & R_{req} \subseteq R_2 \wedge \neg\, \mathsf{fail}, \end{array}$$

such that $R_1 \cap R_2 = \emptyset$ and $R_1 \cup R_2 \subseteq R_{av}$, then

$$\mathcal{L}[\![\mathcal{A}_1 \mid \mathcal{A}_2 \mid \phi]\!] \;\Rightarrow\; \neg\, \mathsf{fail}.$$

2.3 Specification macros

Earlier we introduced the write and stable macros. In this section we present some more specification macros for helping in structuring specifications. The idea is that the macros are quite close to natural language, and so are easily understood, but are completely formal. The macros have many similarities with the specification macros for Timed CSP, presented in [Dav93].

We will often want to say that a particular shunt s is written to at some time t: we write $s \;\mathsf{at}\; t$ to specify this, which is formally defined by:

$$s \;\mathsf{at}\; t \;\widehat{=}\; s@t.ts = t.$$

We can generalize this to specify that s is written to at some time during interval I:

$$s \;\mathsf{at}\; I \;\widehat{=}\; \exists\, t \in I \bullet s \;\mathsf{at}\; t.$$

We will also want to be able to talk about the sequence of values written to s, which we define using a sequence comprehension:

$$\mathsf{trace}\, s \;\widehat{=}\; \langle s@t.v \mid t \leftarrow [t_\alpha, t_\omega] \wedge s \;\mathsf{at}\; t\rangle.$$

We generalize this to talk about the values written before some time t:

$$\mathsf{trace}\, s \;\mathsf{before}\; t \;\widehat{=}\; \langle s@t'.v \mid t' \leftarrow [t_\alpha, t_\omega] \wedge t' \leqslant t \wedge s \;\mathsf{at}\; t'\rangle.$$

It will often be useful to make an environmental assumption that inputs to shunts do not occur too frequently. $\mathsf{constancy}(s, t)$ is true if successive writes to s are separated by an interval of at least t:

$$\mathsf{constancy}(s, t) \;\widehat{=}\; s \;\mathsf{at}\; t_1 \wedge s \;\mathsf{at}\; t_2 \;\Rightarrow\; t_1 = t_2 \vee |t_1 - t_2| > t.$$

This can be generalized to a set of shunts:

$$\begin{aligned}\mathsf{constancy}(S, t) \;\widehat{=}\; \forall\, s_1, s_2 \in S \bullet s_1 \;\mathsf{at}\; t_1 \wedge s_2 \;\mathsf{at}\; t_2 \Rightarrow s_1 = s_2 \wedge t_1 = t_2 \\ \vee\, |t_1 - t_2| > t.\end{aligned}$$

The following predicate states that the values written on shunt s are the same as those written on s', except they are delayed by up to time d:

$$\begin{aligned} s\; d\text{-}\mathsf{delayed}\; s' \;\widehat{=}\;& s@t_\alpha = s'@t_\alpha \\ & \wedge \forall\, t \in [t_\alpha, t_\omega] \bullet \mathsf{write}(v, s', t) \Rightarrow \mathsf{write}(v, s, [t, t + d]) \\ & \qquad\qquad \wedge\, \mathsf{trace}\, s \;\mathsf{before}\; t \leqslant \mathsf{trace}\, s' \;\mathsf{before}\; t.\end{aligned}$$

Note that

$$\begin{aligned} s\; d\text{-}\mathsf{delayed}\; s' \;\Rightarrow\;& \forall\, t \in [t_\alpha, t_\omega] \bullet \exists\, t' \in [t - d, t] \bullet s@t.v = s'@t'.v \\ & \wedge \forall\, t \in [t_\alpha, t_\omega - d] \bullet \exists\, t' \in [t, t + d] \bullet s'@t.v = s@t'.v.\end{aligned}$$

The following lemma states a few results concerning the $\mathsf{delayed}$ macro.

Lemma 2.3:

1. $s\; 0\text{-}\mathsf{delayed}\; s' \Leftrightarrow \forall\, t \in [t_\alpha, t_\omega] \bullet s@t = s'@t.$
2. If $d \leqslant d'$ then $s'\; d\text{-}\mathsf{delayed}\; s \Rightarrow s'\; d'\text{-}\mathsf{delayed}\; s.$
3. If $\mathsf{constancy}(s, d)$ then

$$\begin{aligned} s'\; d\text{-}\mathsf{delayed}\; s \;\Leftrightarrow\;& s@t_\alpha = s'@t_\alpha \\ & \wedge \forall\, t \in [t_\alpha, t_\omega] \bullet \mathsf{write}(v, s, t) \Rightarrow \mathsf{write}(v, s', [t, t + d]) \\ & \qquad\qquad \wedge\, \mathsf{write}(v, s', t) \Rightarrow \mathsf{write}(v, s, [t - d, t]).\end{aligned}$$

4. $s'\; d\text{-}\mathsf{delayed}\; s \wedge s''\; d'\text{-}\mathsf{delayed}\; s' \Rightarrow s''\; (d + d')\text{-}\mathsf{delayed}\; s.$

3 A technique for writing timed specifications

In this section we develop a technique for specifying systems based upon separating the functional and temporal parts of the specification. We believe that this makes specifying systems easier. We will also present results that help to transform such specifications into code.

We begin with an example. Suppose we want to specify a system that inputs natural numbers on channel *in* and outputs their squares on channel *out* within 5 seconds. Specifying this formally yet concisely is remarkably difficult: many plausible looking specifications prove incorrect under closer examination. For example, one naive attempt at specifying this would be:

$$\forall\, t \in [t_\alpha, t_\omega] \bullet \mathsf{write}(n, in, t) \Rightarrow \mathsf{write}(n^2, out, [t, t+5]).$$

However, this specification allows the agent to generate spurious outputs. An improvement would be to add a conjunct:

$$\forall\, t \in [t_\alpha, t_\omega] \bullet \mathsf{write}(n^2, out, t) \Rightarrow \mathsf{write}(n, in, [t-5, t]).$$

However, this is still not right if two inputs may occur very close in time to one another: the specification allows the corresponding outputs to occur in the opposite order; it also allows two identical inputs to be combined to give a single output.

Our approach will be to consider a virtual (i.e. imaginary) channel out' which always holds the square of the current input, without any delay. Our specification then consists of two parts:

- out' always holds the square of the value on *in*;
- *out* is a delayed copy of out'.

Formally:

$$\exists\, out' \bullet (\forall\, t \in [t_\alpha, t_\omega] \bullet out'@t.v = in@t.v^2) \wedge out\ 5\text{-}\mathsf{delayed}\ out'.$$

This form of specification will be sufficiently useful that we introduce a specification macro for it:

Definition 3.1 (Delay)**:** Let *spec* be a specification over two channels *in* and *out*. Then:

$$\mathsf{Delay}(spec, in, out, d) \ \widehat{=}\ \exists\, out' \bullet spec(in, out') \wedge out\ d\text{-}\mathsf{delayed}\ out'.$$

For example, the above specification can be written as:

$$\begin{array}{l}\mathsf{Delay}(spec, in, out, 5),\\ \text{where } spec(in, out) \mathrel{\widehat{=}} \forall\, t \in [t_\alpha, t_\omega] \bullet out@t.v = in@t.v^2.\end{array}$$

This specification technique allows us to separate the functional and timeliness parts of the specification. It can be thought of as firstly specifying how we would like our system to behave if we had an infinitely fast computer (or equivalently if our time granularity were very large), and then introducing the possibility of delays. In the above example, we would ideally like inputs to be squared and output immediately, as specified by *spec*; but this is unfeasible so we allow a delay of up to 5 seconds, introduced by the Delay macro.

The following lemma allows us to decrease delays, or to strengthen the delayed specification:

Lemma 3.2 (Monotonicity of Delay): If $d \leqslant d'$ and $spec \Rightarrow spec'$ then

$$\mathsf{Delay}(spec, in, out, d) \ \Rightarrow\ \mathsf{Delay}(spec', in, out, d').$$

The following theorem allows two delayed specifications to be chained together:

Theorem 3.3 (Chaining of Delays): If $\hat{d}$ is such that

$$\begin{array}{l} spec_1(in, mid') \wedge spec_2(mid, out) \wedge mid\ d\text{-}\mathsf{delayed}\ mid' \Rightarrow \\ \quad \exists\, out' \bullet spec_2(mid', out') \wedge out\ \hat{d}\text{-}\mathsf{delayed}\ out', \end{array} \tag{1}$$

then

$$\begin{array}{l} \mathsf{Delay}(spec_1, in, mid, d) \wedge \mathsf{Delay}(spec_2, mid, out, d') \Rightarrow \\ \quad \mathsf{Delay}(spec_1 \mathbin{;} spec_2, in, out, \hat{d} + d'), \end{array}$$

where

$$(spec_1 \mathbin{;} spec_2)(in, out) \mathrel{\widehat{=}} \exists\, mid \bullet spec_1(in, mid) \wedge spec_2(mid, out).$$

The condition (1) says that if the shunts in and mid' satisfy $spec_1$, and mid and out satisfy $spec_2$, and mid is a d-delayed copy of mid', then out is a $\hat{d}$-delayed version of some shunt out' such that mid' and out' satisfy $spec_2$. We will normally take $\hat{d} = d$, but sometimes we will be able to take $\hat{d} = 0$, or some other value. This theorem follows from the more general rule for chaining Delays, given below.

We can generalize the Delay macro to more than one input; for ease of notation, we write the inputs as a vector:

$$\begin{array}{rl} \mathsf{Delay}(spec, \langle in_1, \ldots, in_n \rangle, out, d) \mathrel{\widehat{=}} & \exists\, out' \bullet spec(\langle in_1, \ldots, in_n \rangle, out') \\ & \wedge\ out\ d\text{-}\mathsf{delayed}\ out'. \end{array}$$

Here $spec$ takes n inputs and gives one output.

Henceforth, we will write $\underline{in}$ as shorthand for $in_1, \ldots, in_n$, etc.

Suppose we have a vector of input shunts $\underline{in}$, a vector of intermediate shunts $\underline{mid}$, and an output shunt out, such that: each intermediate shunt mid_i is related to the inputs by $spec_i$, delayed by d_i; and the output is related to the intermediate values by $spec$, delayed by d. The following theorem relates the output to the inputs directly.

Theorem 3.4 (Chaining of Delays): If $\hat{d}$ is such that

$$\begin{array}{l} \big(spec(\underline{mid}, out) \wedge \forall\, i : 1 \,..\, n \bullet spec_i(\underline{in}, mid_i') \wedge mid_i\ d_i\text{-}\mathsf{delayed}\ mid_i'\big) \Rightarrow \\ \quad \exists\, out' \bullet spec(\underline{mid}', out') \wedge out\ \hat{d}\text{-}\mathsf{delayed}\ out', \end{array} \tag{2}$$

then

$$\begin{array}{l} \big(\forall\, i : 1 \,..\, n \bullet \mathsf{Delay}(spec_i, \underline{in}, mid_i, d_i) \wedge \mathsf{Delay}(spec, \underline{mid}, out, d)\big) \Rightarrow \\ \quad \mathsf{Delay}(\widehat{spec}, \underline{in}, out, d + \hat{d}), \end{array}$$

where

$$(\widehat{spec})(\underline{in}, out) \mathrel{\widehat{=}} \exists\, \underline{mid} \bullet (\forall\, i : 1 \,..\, n \bullet spec_i(\underline{in}, mid_i)) \wedge spec(\underline{mid}, out).$$

We will often take $\hat{d} = \max\{d_i \mid i \in 1 \,..\, n\}$. The proof appears in [LZ95].

Note that in the above we have assumed that each intermediate value mid_i is related to *all* the inputs by some specification $spec_i$; in practice, $spec_i$ will mention only some of the inputs, but we can still use the above theorem by implicitly adding extra arguments to $spec_i$.

Note also that the theorem assumes that there is a specification $spec_i$ for *every* mid_i; in practice, some of the mid_is will themselves be inputs, and used to calculate some other intermediate values. When this happens, we can still use the theorem: if s is such a shunt, used both as an input and an intermediate value, then we treat s as an input, and introduce another, intermediate, shunt s' such that $\mathsf{Delay}(EQUAL, s, s', 0)$ where $EQUAL(s, s') \mathrel{\widehat{=}} \forall\, t \bullet s@t = s'@t$. We then replace references to s in $spec$ by s'. But from the above, we know that $\forall\, t \bullet s@t = s'@t$, so we can treat s and s' as equal. We will not bother to explicitly state when we use this trick.

4 The case study

Our case study concerns the software requirements of the A-7E aircraft's Operational Flight Program. This system exhibits hard real-time behaviour in processing of pilot data entry, and the control of a screen and confirmation lights. It has already been studied in detail, and we are therefore able to compare our results with alternative solutions. We take our informal requirements from Heitmeyer and Labaw's specification [HL87] in RTL [JM86].

The Operational Flight Program is concerned with five elements: an alphanumeric keyboard, which is used for data entry; a data entry button, which is used by the pilot to signal that he is about to start entering data; a data accept button, used to signal the end of data entry; a screen, on which input is echoed; and a confirmation light, used to signal to the pilot that he may input data.

The pilot is required to follow a strict protocol, designed to reduce the risk of invalid or spurious data entry. Firstly, he must press the data entry button; the system will then respond by turning on the confirmation light and clearing the screen. The pilot may now enter the data, which will appear on the screen as he types. When the pilot has completed entry, he must read it before pressing the data accept button, at which stage the confirmation light will go out. If the pilot does not leave a sufficient pause between typing the last key and pressing the accept button (at least time t_{acpt}) then pressing the accept button is invalid, and will be ignored.

4.1 Formal specification

We will model each of the inputs and outputs as shunts. For the data entry button, accept button and screen clear, we are not interested in the values passed, only their time stamps: we therefore assume that they are of type *Unit*, a type containing only one value, *nil*.

We have three input shunts: the data entry button, the accept button and the keyboard:

$$\begin{array}{rcll} DEB & : & Time \times Unit & \text{(data entry button)}, \\ ACPT & : & Time \times Unit & \text{(accept button)}, \\ KYBD & : & Time \times Key & \text{(keyboard)}. \end{array}$$

We do not further specify the type *Key* of keyboard entries, except that it is finite. We have three output shunts: the confirmation light, the display, and the screen clear:

$$\begin{array}{rcll} CONF & : & Time \times Bool & \text{(confirmation light)}, \\ DSPLY & : & Time \times Key & \text{(display)}, \\ SCLR & : & Time \times Unit & \text{(screen clear)}. \end{array}$$

We now formalize our specification. We assume that the informal requirements are to be met after some warming-up period of length t_{init}. We would like the requirements to be met for ever, but the models of TAM deal only with finite executions, so we will specify that the requirements are to be met until some time T, which we can take to be arbitrarily large. Hence each specification will include a quantification "$\forall\, t \in [t_\alpha + t_{init}, T]$".

We define a predicate that indicates whether the system is currently willing to receive keyboard input. This is true if there is some time t' in the past at which the data entry button was pressed, such that the accept button has not been validly pressed since t' (a press of the accept button is invalid if it occurs within t_{acpt} of the previous key press, i.e. if insufficient time was allowed for checking the data):

$$\begin{aligned}
&acc(DEB, ACPT, KYBD)(t) \mathrel{\widehat=} \\
&\quad \exists\, t' \leqslant t \bullet DEB \text{ at } t' \\
&\qquad\qquad \wedge \nexists\, t'' \in (t', t] \bullet ACPT \text{ at } t'' \wedge \neg\, KYBD \text{ at } [t'' - t_{acpt}, t''].
\end{aligned}$$

We will choose to drop the arguments $DEB, ACPT, KYBD$, and simply write $acc(t)$, where this will not cause confusion.

Valid key presses should be echoed on the screen within t_{dsply}:

$$\begin{aligned}
&R_1(\langle DEB, ACPT, KYBD\rangle, DSPLY) \mathrel{\widehat=} \\
&\quad \forall\, t \in [t_\alpha + t_{init}, T] \bullet \mathsf{write}(v, KYBD, t) \wedge acc(t) \Leftrightarrow \mathsf{write}(v, DSPLY, t), \\
&TR_1 \mathrel{\widehat=} \mathsf{Delay}(R_1, \langle DEB, ACPT, KYBD\rangle, DSPLY, t_{dsply}).
\end{aligned}$$

R_1 specifies that the key presses are echoed immediately; TR_1 then introduces a delay of up to t_{dsply}. Here and henceforth we adopt the convention of writing R_1, etc. for "untimed" specifications, and TR_1, etc. for the corresponding timed specifications. Note that R_1 is a function of the shunts DEB, $ACPT$ and $KYBD$ because acc is.

The keyboard light should be turned off within t_{conf} of leaving accepting mode:

$$\begin{aligned}
&R_2(\langle DEB, ACPT, KYBD\rangle, CONF) \mathrel{\widehat=} \\
&\quad \forall\, t \in [t_\alpha + t_{init}, T] \bullet \neg\, acc(t) \wedge acc(t-1) \Leftrightarrow \mathsf{write}(false, CONF, t), \\
&TR_2 \mathrel{\widehat=} \mathsf{Delay}(R_2, \langle DEB, ACPT, KYBD\rangle, CONF, t_{conf}).
\end{aligned}$$

The keyboard light should be turned on within t_{conf} of entering accepting mode:

$$\begin{aligned}
&R_3(\langle DEB, ACPT, KYBD\rangle, CONF) \mathrel{\widehat=} \\
&\quad \forall\, t \in [t_\alpha + t_{init}, T] \bullet acc(t) \wedge \neg\, acc(t-1) \Leftrightarrow \mathsf{write}(true, CONF, t), \\
&TR_3 \mathrel{\widehat=} \mathsf{Delay}(R_3, \langle DEB, ACPT, KYBD\rangle, CONF, t_{conf}).
\end{aligned}$$

The screen should be cleared within t_{sclr} of entering accepting mode:

$$\begin{aligned}
&R_4(\langle DEB, ACPT, KYBD\rangle, SCLR) \mathrel{\widehat=} \\
&\quad \forall\, t \in [t_\alpha + t_{init}, T] \bullet acc(t) \wedge \neg\, acc(t-1) \Leftrightarrow SCLR \text{ at } t, \\
&TR_4 \mathrel{\widehat=} \mathsf{Delay}(R_4, \langle DEB, ACPT, KYBD\rangle, SCLR, t_{sclr}).
\end{aligned}$$

In order to achieve the above requirements, it will be necessary to make an assumption about the environment in which it operates, namely that inputs occur with frequency no greater than once every t_{kmin}:

$$EA \mathrel{\widehat=} \mathsf{constancy}(\{DEB, ACPT, KYBD\}, t_{kmin}).$$

We assume that the time parameter t_{acpt} is specified by the customer, but he does not specify the values t_{init}, t_{dsply}, t_{conf}, t_{sclr} and t_{kmin}, except that he wants them as small as possible: during the development of the system, we will calculate values for the time parameters; the customer will then have to decide whether these times are acceptable to him.

We can now write our complete specification. We may alter the values held on the output shunts, and, assuming the environmental assumption holds, we must ensure TR_1 to TR_4 are met:

$$SPEC \mathrel{\widehat=} \{DSPLY, CONF, SCLR\} : EA \Rightarrow TR_1 \wedge TR_2 \wedge TR_3 \wedge TR_4.$$

4.2 Decomposition

The quantity $acc(t)$ is of central importance, so it seems sensible to introduce a shunt ACC, which will hold its (boolean) value:

$$\begin{array}{ll} & SPEC \\ \sqsubseteq & \langle \text{introduce variable (law A.3)} \rangle \\ & \mathsf{shunt}\, ACC : Time \times Bool\ \mathsf{in} \\ & \{DSPL, CONF, SCLR, ACC\} : EA \Rightarrow TR_1 \wedge TR_2 \wedge TR_3 \wedge TR_4 . \qquad \lhd \end{array}$$

Here and henceforth, following Morgan's conventions [Mor90], we use the symbol $\lhd$ to mark the part of the program that we will continue to refine; the rest of the program (in the above, the "$\mathsf{shunt}\, ACC\ \mathsf{in}$" part) is implicitly carried forward around it. If we want to refine parts of the program separately, we will mark those parts (i), (ii), etc.

We will strengthen the specification to insist that the shunt ACC holds the correct value; however, we will, of course, have some delay in calculating this value, so we write a delayed specification:

$$\begin{array}{l} R_{acc}(\langle DEB, ACPT, KYBD \rangle, ACC) \mathrel{\widehat{=}} \\ \qquad \forall\, t \in [t_\alpha + t_{init}, T] \bullet ACC@t.v = acc(DEB, ACPT, KYBD)(t), \\ TR_{acc} \mathrel{\widehat{=}} \mathsf{Delay}(R_{acc}, \langle DEB, ACPT, KYBD \rangle, ACC, t_{acc}). \end{array}$$

We rewrite the untimed requirements R_1–R_4 to use the value of ACC:

$$\begin{array}{l} R'_1(\langle ACC, KYBD \rangle, DSPLY) \mathrel{\widehat{=}} \\ \qquad \forall\, t \in [t_\alpha + t_{init}, T] \bullet \\ \qquad\qquad \mathsf{write}(v, KYBD, t) \wedge ACC@t.v \Leftrightarrow \mathsf{write}(v, DSPLY, t), \\ R'_2(\langle ACC \rangle, CONF) \mathrel{\widehat{=}} \\ \qquad \forall\, t \in [t_\alpha + t_{init}, T] \bullet \\ \qquad\qquad \neg\, ACC@t.v \wedge ACC@(t-1).v \Leftrightarrow \mathsf{write}(false, CONF, t), \\ R'_3(\langle ACC \rangle, CONF) \mathrel{\widehat{=}} \\ \qquad \forall\, t \in [t_\alpha + t_{init}, T] \bullet \\ \qquad\qquad ACC@t.v \wedge \neg\, ACC@(t-1).v \Leftrightarrow \mathsf{write}(true, CONF, t), \\ R'_4(\langle ACC \rangle, SCLR) \mathrel{\widehat{=}} \\ \qquad \forall\, t \in [t_\alpha + t_{init}, T] \bullet ACC@t.v \wedge \neg\, ACC@(t-1).v \Leftrightarrow SCLR\ \mathsf{at}\ t. \end{array}$$

And we lift these to timed specifications:

$$\begin{array}{lcl} TR'_1 & \mathrel{\widehat{=}} & \mathsf{Delay}(R'_1, \langle ACC, KYBD \rangle, DSPLY, t'_{dsply}), \\ TR'_2 & \mathrel{\widehat{=}} & \mathsf{Delay}(R'_2, \langle ACC \rangle, CONF, t'_{conf}), \\ TR'_3 & \mathrel{\widehat{=}} & \mathsf{Delay}(R'_3, \langle ACC \rangle, CONF, t'_{conf}), \\ TR'_4 & \mathrel{\widehat{=}} & \mathsf{Delay}(R'_4, \langle ACC \rangle, SCLR, t'_{sclr}). \end{array}$$

where t'_{dsply}, t'_{conf} and t'_{sclr} are new time parameters, for which we will calculate values below.

We refine our specification to use these:

$$\begin{array}{ll} \sqsubseteq & \{DSPLY, CONF, SCLR, ACC\} : \\ & \qquad EA \Rightarrow TR_{acc} \wedge TR'_1 \wedge TR'_2 \wedge TR'_3 \wedge TR'_4 . \end{array}$$

In order to verify this refinement step (using law A.1), we must prove:

$$EA \wedge TR_{acc} \wedge TR'_1 \wedge TR'_2 \wedge TR'_3 \wedge TR'_4 \ \Rightarrow\ TR_1 \wedge TR_2 \wedge TR_3 \wedge TR_4 .$$

This can be proved using theorem 3.4. The proof is omitted due to lack of space: the interested reader is referred to [LZ95]. The following conditions arise during the proof:

$$\begin{aligned} t'_{dsply} &\leqslant t_{dsply}, & t'_{conf} + t_{acc} &\leqslant t_{conf}, \\ t'_{sclr} + t_{acc} &\leqslant t_{sclr}, & t_{acc} &\leqslant t_{kmin}. \end{aligned}$$

We now split the specification into four components, one controlling each of the four shunts in the frame (using law A.5):

$$\begin{array}{ll} \sqsubseteq & \{DSPLY\} : (EA \Rightarrow TR'_1) \\ \mid & \{CONF\} : (EA \Rightarrow TR'_2 \wedge TR'_3) \\ \mid & \{SCLR\} : (EA \Rightarrow TR'_4) \\ \mid & \{ACC\} : (EA \Rightarrow TR_{acc}). \end{array}$$

4.3 Design of components

We now exploit some symmetry between the conditions we have to ensure. Each of the four conditions R'_1, $R'_2 \wedge R'_3$, R'_4 and R_{acc} satisfies an equation of the form:

$$R(\underline{in}, out) \;\Leftarrow\; \begin{pmatrix} \forall\, m : 1 \,..\, M \bullet \forall\, t \in [t_\alpha + t_{init}, T] \bullet \\ C_m(\underline{in}, t) \Leftrightarrow \mathsf{write}(v_m, out, t) \end{pmatrix}, \tag{3}$$

for some disjoint conditions $C_1, \ldots, C_M$. For example, if we take $Key = \{k_m \mid m \in 1 \,..\, M\}$, then $R'_1(\langle ACC, KYBD \rangle, DSPLY)$ can be written as:

$$\begin{array}{l} \forall\, m : 1 \,..\, M \bullet \forall\, t \in [t_\alpha + t_{init}, T] \bullet \\ \quad KYBD@t = (t, k_m) \wedge ACC@t.v \Leftrightarrow \mathsf{write}(k_m, DSPLY, t). \end{array}$$

Also, there are related conditions C'_m such that by testing the C'_m periodically, we can tell whether C_m were true at some intermediate time: for any times $t_0, \ldots, t_n$ such that $t_i \in (t_\alpha + i.p, t_\alpha + (i+1)p]$ (for suitable n and p), C_m is true at some time between t_{i-1} and t_i iff C'_m is true when applied to the values of the inputs at t_i and t_{i-1}:

$$\begin{array}{ll} \mathsf{if} & \forall\, i : 0 \,..\, n \bullet t_i \in (t_\alpha + i.p, t_\alpha + (i+1)p] \\ \mathsf{then} & \forall\, m : 1 \,..\, M \bullet \forall\, i : 1 \,..\, n \bullet \\ & \quad (\exists\, t \in (t_{i-1}, t_i] \bullet C_m(\underline{in}, t)) \Leftrightarrow C'_m(\underline{in}@t_i, \underline{in}@t_{i-1}, t_i, t_{i-1}). \end{array} \tag{4}$$

For example, for R'_1 we can write C'_m as

$$KYBD@t_i.v = k_m \wedge KYBD@t_i.ts > t_{i-1} \wedge ACC@t_i.v.$$

Further, the inputs vary infrequently:

$$\mathsf{constancy}(\underline{in}, t_{sep}). \tag{5}$$

For example, the inputs $KYBD$ and ACC vary at must every $t_{kmin} - t_{acc}$.

Lastly, the C_m are true only when an input changes, and so outputs must occur infrequently:

$$C_m(\underline{in}, t) \wedge C_{m'}(\underline{in}, t') \;\Rightarrow\; m = m' \wedge t = t' \vee |t - t'| > t_{sep}. \tag{6}$$

We exploit these facts in the following theorem.

Theorem 4.1: If equations 3–6 are satisfied then

$$
\begin{array}{l}
\{out\} : Delay(R, \underline{in}, out, d) \ \sqsubseteq \\
\quad \mathsf{var}\ \underline{current}, \underline{previous}, current\_ts, previous\_ts\ \mathsf{in} \\
\quad [p] \begin{pmatrix} current\_ts := Now \ ; current_1 \leftarrow in_1 \ ; \ldots ; current_k \leftarrow in_k ; \\ current\_ts := \max\{current\_ts, current_1.ts, \ldots, current_k, ts\} \end{pmatrix} ; \\
\quad \mathsf{loop\,for}\ n\ \mathsf{period}\ p \\
\qquad previous_1 := current_1 \ ; \ldots ; previous_k := current_k ; \\
\qquad previous\_ts := current\_ts ; \\
\qquad current\_ts := Now \ ; current_1 \leftarrow in_1 \ ; \ldots ; current_k \leftarrow in_k ; \\
\qquad current\_ts := \max\{current\_ts, current_1.ts, \ldots, current_k, ts\} ; \\
\qquad \mathsf{if} \ \underset{m \in 1..M}{\Box} \begin{pmatrix} C'_m(\underline{current}, \underline{previous}, current\_ts, previous\_ts) \\ \Rightarrow v_m \rightarrow out \end{pmatrix} \ \mathsf{fi} \\
\quad \mathsf{endloop},
\end{array}
$$

provided $t_{sep} \geqslant d$, $t_{init} \geqslant p$, $n.p \geqslant T - t_\alpha$ and $d \geqslant 2.p - 1$, where $\underline{in} = \langle in_1, \ldots, in_k \rangle$, etc.

Proof: We have

$$
\begin{array}{ll}
 & \mathsf{Delay}(R, \underline{in}, out, d) \\
\Leftarrow & \langle \text{definition of Delay; condition 3} \rangle \\
 & \exists\, out' \bullet \forall\, m \bullet \forall\, t \in [t_\alpha + t_{init}, T] \bullet C_m(\underline{in}, t) \Leftrightarrow \mathsf{write}(v_m, out', t) \\
 & \qquad \wedge\ out\ d\text{-}\mathsf{delayed}\ out' \\
\Leftarrow & \langle \text{lemma 2.3 (3), using condition 6 and the condition } t_{sep} \geqslant d \rangle \\
 & \exists\, out' \bullet \forall\, m \bullet \forall\, t \in [t_\alpha + t_{init}, T] \bullet C_m(\underline{in}, t) \Leftrightarrow \mathsf{write}(v_m, out', t) \\
 & \qquad \wedge\ out@0 = out'@0 \\
 & \qquad \wedge\ \forall\, t \in [t_\alpha, T] \bullet \mathsf{write}(v, out', t) \Rightarrow \mathsf{write}(v, out, [t, t + d]) \\
 & \qquad\qquad\qquad\qquad \wedge\ \mathsf{write}(v, out, t) \Rightarrow \mathsf{write}(v, out', [t - d, t]) \\
\Leftarrow & \langle \text{predicate calculus, taking } out' \text{ as defined by the first two lines} \rangle \\
 & \forall\, m \bullet \forall\, t \in [t_\alpha + t_{init}, T] \bullet \\
 & \qquad C_m(\underline{in}, t) \Rightarrow \mathsf{write}(v_m, out, [t, t + d]) \\
 & \qquad \wedge\ \mathsf{write}(v_m, out, t) \Rightarrow \exists\, t' \in [t - d, t] \bullet C_m(\underline{in}, t') \\
\Leftarrow & \langle \text{see below} \rangle \\
 & \neg\ out\ \mathsf{at}\ [t_\alpha, t_\alpha + p] \\
 & \wedge\ \forall\, j : 0 \ldots n \bullet \exists\, t_j \in [t_\alpha + j.p, t_\alpha + (j + 1)p) \bullet \forall\, i : 1 \ldots n \bullet \\
 & \qquad \forall\, m \bullet C'_m(\underline{in}@t_i, \underline{in}@t_{i-1}, t_i, t_{i-1}) \Leftrightarrow \mathsf{write}(v_m, out, [t_i, t_\alpha + (i + 1)p]) \\
 & \qquad \wedge\ \neg\ out\ \mathsf{at}\ (t_\alpha + i.p, t_i].
\end{array}
$$

For the last step, assume the right hand side; we will show the left hand side. Suppose for some $t \in [t_\alpha + t_{init}, T]$, $C_m(\underline{in}, t)$. Then because $t_{init} \geqslant p$ and $n.p \geqslant T - t_\alpha$ we have $t \in (t_{i-1}, t_i]$ for some $i \in 1 \ldots n$. Then from condition 4, $C'_m(\underline{in}@t_i, \underline{in}@t_{i-1}, t_i, t_{i-1})$; so $\mathsf{write}(v_m, out, [t_i, t_\alpha + (i + 1)p])$ from the assumption; hence $\mathsf{write}(v_m, out, [t, t + d])$ because $d \geqslant 2.p - 1$.

Now suppose for some $t \in [t_\alpha + t_{init}, T]$, $\mathsf{write}(v_m, out, t)$. Then as above $t \in (t_{i-1}, t_i]$ for some $i \in 1 \ldots n$. But $\neg\ out\ \mathsf{at}\ (t_\alpha + i.p, t_i]$ so $t \in (t_{i-1}, t_\alpha + i.p]$; also $\neg\ out\ \mathsf{at}\ [t_\alpha, t_\alpha + p]$ so $i > 1$. Hence, taking $j = i - 1$, we have $C'_m(\underline{in}@t_j, \underline{in}@t_{j-1}, t_j, t_{j-1})$, from the assumption; and so from condition 4, there exists some $t' \in (t_{j-1}, t_j]$ such that $C_m(\underline{in}, t')$; but $t' \in [t - d, t]$, because $d \geqslant 2.p - 1$, as required.

Hence we have:

$$
\begin{array}{ll}
 & \{out\} : Delay(spec, \underline{in}, out, d) \\
\sqsubseteq & \langle \text{strengthen specification} \rangle \\
 & \{out\} : \neg\, out \text{ at } [t_\alpha, t_\alpha + p] \\
 & \quad \wedge \forall j : 0 \,..\, n \bullet \exists t_j \in [t_\alpha + j.p, t_\alpha + (j+1)p) \bullet \forall i : 1 \,..\, n \bullet \\
 & \qquad \forall m \bullet C'_m(\underline{in}@t_i, \underline{in}@t_{i-1}, t_i, t_{i-1}) \\
 & \qquad\quad \Leftrightarrow \mathsf{write}(v_m, out, [t_i, t_\alpha + (i+1)p]) \\
 & \qquad \wedge \neg\, out \text{ at } (t_\alpha + i.p, t_i] \\
\sqsubseteq & \langle \text{introduce variables (law A.4); strengthen specification} \rangle \\
 & \mathsf{var}\ \underline{current}, \underline{previous}, current\_ts, previous\_ts\ \mathsf{in} \\
 & \left.\begin{array}{l}
 \{out, \underline{current}, \underline{previous}, current\_ts, previous\_ts\} : \\
 \quad \neg\, out \text{ at } [t_\alpha, t_\alpha + p] \\
 \quad \wedge \forall j : 0 \,..\, n \bullet \exists t_j \in [t_\alpha + j.p, t_\alpha + (j+1)p) \bullet \forall i : 1 \,..\, n \bullet \\
 \qquad \forall m \bullet C'_m(\underline{in}@t_i, \underline{in}@t_{i-1}, t_i, t_{i-1}) \\
 \qquad\quad \Leftrightarrow \mathsf{write}(v_m, out, [t_i, t_\alpha + (i+1)p]) \\
 \qquad \wedge \neg\, out \text{ at } (t_\alpha + i.p, t_i] \\
 \qquad \wedge \underline{current}@(t_\alpha + i.p) = \underline{in}@t_{i-1} \\
 \qquad \wedge current\_ts@(t_\alpha + i.p) = t_{i-1}
 \end{array}\right\} \quad \lhd \\
\sqsubseteq & \langle \text{strengthen specification} \rangle \\
 & \{out, \underline{current}, \underline{previous}, current\_ts, previous\_ts\} : \\
 & \quad \neg\, out \text{ at } [t_\alpha, t_\alpha + p] \\
 & \quad \wedge \exists t_0 \in [t_\alpha, t_\alpha + p) \bullet \underline{current}@(t_\alpha + p) = \underline{in}@t_0 \\
 & \qquad\qquad \wedge current\_ts@(t_\alpha + p) = t_0 \\
 & \quad \wedge \forall i : 1 \,..\, n \bullet \exists t_i \in [t_\alpha + i.p, t_\alpha + (i+1)p) \bullet \\
 & \qquad \forall m \bullet C'_m(\underline{in}@t_i, \underline{current}@(t_\alpha + i.p), t_i, current\_ts@(t_\alpha + i.p)) \\
 & \qquad\quad \Leftrightarrow \mathsf{write}(v_m, out, [t_i, t_\alpha + (i+1)p]) \\
 & \qquad \wedge \neg\, out \text{ at } (t_\alpha + i.p, t_i] \\
 & \qquad \wedge \underline{current}@(t_\alpha + (i+1)p) = \underline{in}@t_i \\
 & \qquad \wedge current\_ts@(t_\alpha + (i+1).p) = t_i \\
\sqsubseteq & \left\langle \begin{array}{l} \text{introduce sequential composition (law A.6); contract frame (law A.2);} \\ \text{introduce loop (law A.7)} \end{array} \right\rangle \\
 & \left.\begin{array}{l}
 \{\underline{current}, current\_ts\} : t_\omega = t_\alpha + p \\
 \qquad \wedge \exists t_0 \in [t_\alpha, t_\omega) \bullet \underline{current}@t_\omega = \underline{in}@t_0 \\
 \qquad\qquad \wedge current\_ts@t_\omega = t_0;
 \end{array}\right\} \quad \text{(i)} \\
 & \mathsf{loop\ for}\ n\ \mathsf{period}\ p \\
 & \quad \left.\begin{array}{l}
 \{out, \underline{current}, \underline{previous}, current\_ts, previous\_ts\} : \\
 \quad \exists t \in [t_\alpha, t_\omega) \bullet \\
 \qquad \forall m \bullet C'_m(\underline{in}@t, \underline{current}@t_\alpha, t, current\_ts@t_\alpha) \\
 \qquad\quad \Leftrightarrow \mathsf{write}(v_m, out, [t, t_\omega]) \\
 \qquad \wedge \neg\, out \text{ at } (t_\alpha, t] \\
 \qquad \wedge \underline{current}@t_\omega = \underline{in}@t \wedge current\_ts@t_\omega = t
 \end{array}\right\} \quad \lhd \\
 & \mathsf{endloop} \\
\sqsubseteq & \langle \text{introduce sequential composition (law A.6)} \rangle \\
 & \left.\begin{array}{l}
 \{\underline{previous}, previous\_ts\} : \underline{previous}@t_\omega = \underline{current}@t_\alpha \\
 \qquad \wedge previous\_ts@t_\omega = current\_ts@t_\alpha;
 \end{array}\right\} \quad \text{(ii)} \\
 & \left.\begin{array}{l}
 \{\underline{current}, current\_ts\} : \exists t \in [t_\alpha, t_\omega) \bullet \underline{current}@t_\omega = in@t \\
 \qquad \wedge current\_ts@t_\omega = t;
 \end{array}\right\} \quad \text{(iii)} \\
 & \left.\begin{array}{l}
 \{out\} : \forall m \bullet C'_m(\underline{current}@t_\alpha, \underline{previous}@t_\alpha, \\
 \qquad current\_ts@t_\alpha, previous\_ts@t_\alpha) \\
 \qquad \Leftrightarrow \mathsf{write}(v_m, out, [t_\alpha, t_\omega])
 \end{array}\right\} \quad \lhd \\
\sqsubseteq & \langle \text{introduce alternation (law A.8)} \rangle
\end{array}
$$

$$\text{if} \bigsqcap_{m \in 1..M} \left(\begin{array}{l} C'_m(\underline{current}, \underline{previous}, current\_ts, previous\_ts) \\ \quad \Rightarrow \{out\} : \mathsf{write}(v_m, out, [t_\alpha, t_\omega]) \end{array} \right) \text{fi}$$

$$\sqsubseteq \langle \text{introduce shunt write (law A.10)} \rangle$$

$$\text{if} \bigsqcap_{m \in 1..M} \left(\begin{array}{l} C'_m(\underline{current}, \underline{previous}, current\_ts, previous\_ts) \\ \quad \Rightarrow v_m \to out \end{array} \right) \text{fi.}$$

The final step is to refine (i)–(iii) above using laws A.6, A.9, A.11 and A.12:

$$\begin{array}{lll} \text{(i)} & \sqsubseteq & [p] \left(\begin{array}{l} current\_ts := Now \,;\, current_1 \leftarrow in_1 \,;\, \ldots \,;\, current_k \leftarrow in_k; \\ current\_ts := \max\{current\_ts, current_1.ts, \ldots, current_k, ts\} \end{array} \right), \\ \text{(ii)} & \sqsubseteq & previous_1 := current_1 \,;\, \ldots \,;\, previous_k := current_k; \\ & & previous\_ts := current\_ts, \\ \text{(iii)} & \sqsubseteq & current\_ts := Now \,;\, current_1 \leftarrow in_1 \,;\, \ldots \,;\, current_k \leftarrow in_k \\ & & current\_ts := \max\{current\_ts, current_1.ts, \ldots, current_k, ts\}. \end{array}$$

Note that the shunt reads are correct even if one of the input shunts is written to sometime in the middle: by assumption (condition 5), at most one shunt is written to; if a shunt is written to at some time t before it is read, then each $current_i$ ends up holding the value of $in_i@t$, and $current\_ts$ ends up holding t; if no shunt is written to before it is read then each $current_i$ ends up holding the initial value of the corresponding in_i, and $current\_ts$ ends up holding a corresponding time.

Collecting code gives us the desired program. □

We now apply the above theorem to each of the components of the system.

4.3.1 Display

As explained above, we assume that the set of keyboard inputs is given by $Key = \{k_m \mid m \in 1 \,..\, M\}$, and define

$$\begin{array}{rcl} C_m(KYBD, ACC, t) & \hat{=} & KYBD@t = (t, v_m) \wedge ACC@t.v, \\ v_m & \hat{=} & k_m, \\ \multicolumn{3}{l}{C'_m(KYBD@t_i, ACC@t_i, KYBD@t_{i-1}, ACC@t_{i-1}, t_i, t_{i-1}) \hat{=}} \\ \multicolumn{3}{l}{\qquad KYBD@t_i.v = v_m \wedge KYBD@t_i.ts > t_{i-1} \wedge ACC@t_i.v.} \end{array}$$

It is easily seen that these definitions satisfy the conditions 3–6 if we take $t_{sep} \hat{=} t_{kmin} - t_{acc}$, so by theorem 4.1:

$$\{DSPLY\} : EA \Rightarrow TR'_1 \sqsubseteq$$

$$\begin{array}{l} \mathsf{var}\ current\_kybd, current\_acc, previous\_kybd, \\ \quad previous\_acc, current\_ts, previous\_ts\ \mathsf{in} \\ [p] \left(\begin{array}{l} current\_ts := Now; \\ current\_kybd \leftarrow KYBD \,;\, current\_acc \leftarrow ACC; \\ current\_ts := \max\{current\_ts, current\_kybd.ts, current\_acc.ts\} \end{array} \right); \\ \mathsf{loop\,for}\ n\ \mathsf{period}\ p \\ \quad previous\_kybd := current\_kybd \,;\, previous\_acc := current\_acc; \\ \quad previous\_ts := current\_ts; \\ \quad current\_ts := Now \,;\, current\_kybd \leftarrow KYBD \,;\, current\_acc \leftarrow ACC; \\ \quad current\_ts := \max\{current\_ts, current\_kybd.ts, current\_acc.ts\}; \\ \quad \mathsf{if} \bigsqcap_{m \in 1..M} \left(\begin{array}{l} current\_kybd.v = v_m \\ \wedge\ current\_kybd.ts > previous\_ts \\ \wedge\ current\_acc.v \end{array} \right) \Rightarrow v_m \to out\ \mathsf{fi} \\ \mathsf{endloop.} \end{array}$$

This program can be further refined in two ways. Firstly, the indexed alternation can obviously be simplified to the single test:

$$\text{if } \begin{pmatrix} current\_kybd.ts > previous\_ts \\ \wedge\ current\_acc.v \end{pmatrix} \Rightarrow current\_kybd.v \rightarrow out \ \text{ fi.}$$

Secondly, the variables *previous_kybd* and *previous_acc* are never used so can be removed from the program:

$\sqsubseteq$ *Display_Manager*,

where

$Display\_Manager \mathrel{\widehat{=}}$

$$\begin{array}{l}
\text{var } current_kybd, current_acc, current_ts, previous_ts \text{ in} \\
[p] \begin{pmatrix} current_ts := Now \,;\, current_kybd \leftarrow KYBD \,;\, current_acc \leftarrow ACC; \\ current_ts := \max\{current_ts, current_kybd.ts, current_acc.ts\} \end{pmatrix}; \\
\text{loop for } n \text{ period } p \\
\quad previous_ts := current_ts; \\
\quad current_ts := Now \,;\, current_kybd \leftarrow KYBD \,;\, current_acc \leftarrow ACC; \\
\quad current_ts := \max\{current_ts, current_kybd.ts, current_acc.ts\}; \\
\quad \text{if } \begin{pmatrix} current_kybd.ts > previous_ts \\ \wedge\ current_acc.v \end{pmatrix} \Rightarrow current_kybd.v \rightarrow DSPLY \ \text{ fi} \\
\text{endloop.}
\end{array}$$

4.3.2 Confirmation light

We define:

$$\begin{aligned}
C_1(ACC, t) &\mathrel{\widehat{=}} \neg\, ACC@t.v \wedge ACC@(t-1).v, \\
C_2(ACC, t) &\mathrel{\widehat{=}} ACC@t.v \wedge \neg\, ACC@(t-1).v, \\
v_1 &\mathrel{\widehat{=}} false, \\
v_2 &\mathrel{\widehat{=}} true, \\
C_1'(ACC@t_i, ACC@t_{i-1}, t_i, t_{i-1}) &\mathrel{\widehat{=}} \neg\, ACC@t_i.v \wedge ACC@t_{i-1}.v, \\
C_2'(ACC@t_i, ACC@t_{i-1}, t_i, t_{i-1}) &\mathrel{\widehat{=}} ACC@t_i.v \wedge \neg\, ACC@t_{i-1}.v.
\end{aligned}$$

The conditions of theorem 4.1 are satisfied if we take $t_{sep} \mathrel{\widehat{=}} t_{kmin} - t_{acc}$. The resulting program does not use the variables *current_ts* and *previous_ts*, so we may remove them to give:

$$\{CONF\} : (EA \Rightarrow TR_2' \wedge TR_3') \sqsubseteq Confirmation\_Light\_Manager,$$

where

$Confirmation\_Light\_Manager \mathrel{\widehat{=}}$

$$\begin{array}{l}
\text{var } current_acc, previous_acc \text{ in} \\
[p]\ current_acc \leftarrow ACC; \\
\text{loop for } n \text{ period } p \\
\quad previous_acc := current_acc; \\
\quad current_acc \leftarrow ACC; \\
\quad \text{if } \ \neg\, current_acc.v \wedge previous_acc.v \Rightarrow false \rightarrow CONF \\
\quad \Box \ \ current_acc.v \wedge \neg\, previous_acc.v \Rightarrow true \rightarrow CONF \\
\quad \text{fi} \\
\text{endloop.}
\end{array}$$

4.3.3 Screen clear

This is very similar to the previous case. We define:

$$C_1(ACC, t) \mathrel{\widehat{=}} ACC@t.v \wedge \neg\, ACC@(t-1).v,$$

$$v_1 \;\widehat{=}\; nil,$$
$$C_1'(ACC@t_i, ACC@t_{i-1}, t_i, t_{i-1}) \;\widehat{=}\; ACC@t_i.v \wedge \neg\, ACC@t_{i-1}.v.$$

And:

$\{CONF\} : (EA \Rightarrow TR_2' \wedge TR_3') \sqsubseteq Screen\_Clear\_Manager,$

where

$Screen\_Clear\_Manager \;\widehat{=}$
  var $current\_acc, previous\_acc$ in
  $[p]$ $current\_acc \leftarrow ACC;$
  loop for n period p
    $previous\_acc := current\_acc;$
    $current\_acc \leftarrow ACC;$
    if $current\_acc.v \wedge \neg\, previous\_acc.v \Rightarrow nil \rightarrow CONF$ fi
  endloop.

4.3.4 Accepting mode manager

We define:

$$\begin{aligned}
C_1(DEB, ACPT, KYBD, t) &\;\widehat{=}\; DEB \text{ at } t\\
C_2(DEB, ACPT, KYBD, t) &\;\widehat{=}\; ACPT \text{ at } t \wedge KYBD@t.ts < t - t_{acpt}\\
v_1 &\;\widehat{=}\; true\\
v_2 &\;\widehat{=}\; false\\
C_1'\begin{pmatrix} DEB@t_i, ACPT@t_i, KYBD@t_i, DEB@t_{i-1},\\ ACPT@t_{i-1}, KYBD@t_{i-1}, t_i, t_{i-1}\end{pmatrix} &\;\widehat{=}\; DEB@t_i.ts > t_{i-1}\\
C_2'\begin{pmatrix} DEB@t_i, ACPT@t_i, KYBD@t_i, DEB@t_{i-1},\\ ACPT@t_{i-1}, KYBD@t_{i-1}, t_i, t_{i-1}\end{pmatrix} &\;\widehat{=}\\
ACPT@t_i.ts > t_{i-1} \wedge KYBD@t_i.ts &< ACPT@t_i.ts - t_{acpt},
\end{aligned}$$

to give

$\{ACC\} : TR_{acc}' \sqsubseteq Accepting\_Mode\_Manager,$

where

$Accepting\_Mode\_Manager \;\widehat{=}$
  var $current\_deb, current\_acpt, current\_kybd, current\_ts, previous\_ts$ in
  $$[p] \begin{pmatrix} current\_ts := Now \,;\, current\_deb \leftarrow DEB;\\ current\_acpt \leftarrow ACPT \,;\, current\_kybd \leftarrow KYBD;\\ current\_ts := \max\{current\_ts, current\_deb.ts,\\ \qquad current\_acpt.ts, current\_kybd.ts\}\end{pmatrix};$$
  loop for n period p
    $previous\_ts := current\_ts;$
    $current\_ts := Now \,;\, current\_deb \leftarrow DEB;$
    $current\_acpt \leftarrow ACPT \,;\, current\_kybd \leftarrow KYBD;$
    $current\_ts := \max\{current\_ts, current\_deb.ts,$
      $current\_acpt.ts, current\_kybd.ts\};$
    if $current\_deb.ts > previous\_ts \Rightarrow true \rightarrow ACC$
    $\Box$ $\begin{pmatrix} current\_acpt.ts > t_{i-1}\\ \wedge\; current\_kybd.ts < current\_acpt.ts - t_{acpt}\end{pmatrix} \Rightarrow false \rightarrow ACC$
    fi
  endloop.

4.4 Scheduling

We now consider the problem of scheduling our program. We assume that we have access to a single CPU r:

$$R_{av} \supseteq Time \times \{r\}.$$

Before we can design a scheduler, we need to know how long the CPU takes to perform each atomic command and evaluate each guard; in order to keep the numbers fairly simple, we assume that the CPU can perform each atomic operation in one time unit:

$$\forall L \in Label\ ;\ t \in Time \bullet \#\phi(t, L) = 1.$$

Formally, our requirements can be written as:

$$SPEC \;\widehat{=}\; \{DSPL, CONF, SCLR\} : EA \Rightarrow TR_1 \wedge TR_2 \wedge TR_3 \wedge TR_4.$$

Assuming the environmental assumption, we must ensure TR_1 to TR_4, and the agent must not fail. From the above, we know that:

$$\begin{aligned} SPEC \;\sqsubseteq_{\mathcal{F}}\; & Display\_Manager \mid Confirmation\_Light\_Manager \mid \\ & Screen\_Clear\_Manager \mid Accepting\_Mode\_Manager. \end{aligned}$$

So we may use theorem 2.1 to deduce that if

$$\begin{aligned} Display\_Manager' &\ labels\ Display\_Manager, \\ Confirmation\_Light\_Manager' &\ labels\ Confirmation\_Light\_Manager, \\ Screen\_Clear\_Manager' &\ labels\ Screen\_Clear\_Manager, \\ Accepting\_Mode\_Manager' &\ labels\ Accepting\_Mode\_Manager, \end{aligned}$$

and

$$\mathcal{L}\left[\!\!\left[\begin{array}{l} Display\_Manager' \mid Confirmation\_Light\_Manager' \mid \\ Screen\_Clear\_Manager' \mid Accepting\_Mode\_Manager' \mid \phi \end{array}\right]\!\!\right] \Rightarrow \neg\, \mathsf{fail},$$

then

$$\begin{aligned} SPEC \;\sqsubseteq_{\mathcal{L}}\; & Display\_Manager' \mid Confirmation\_Light\_Manager' \mid \\ & Screen\_Clear\_Manager' \mid Accepting\_Mode\_Manager' \mid \phi. \end{aligned}$$

In designing the scheduler ϕ, we will need to make design decisions depending upon the values given to the time parameters t_{dsply}, t_{conf} and t_{sclr}. In practice, our customer will want our system to make these parameters as small as possible, but will have some priority between them. In the rest of this section we assume that our customer is particularly keen for keyboard input to be promptly echoed on the screen, i.e. he is keen to see t_{dsply} minimised. We will therefore take the period on the main loop for the display component to be fairly small. It seems important to also have a short period for the accepting mode manager, because delays here cause delays throughout the system.

We make the design decision to give the display and accepting mode managers periods of 21 time units, and the confirmation light and screen clear managers periods of 42 time units. By counting the number of atomic commands in each component, we see that this will require 41 time units of CPU every 42 time units, so this strategy appears likely to succeed. We need to split up the CPU time between the components, which we do as follows:

Component	Resources available
$Display\_Manager'$	$\{(t, r) \mid t \bmod 21 \in [9, 16)\}$
$Confirmation\_Light\_Manager'$	$\{(t, r) \mid t \bmod 42 \in [16, 21)\}$
$Screen\_Clear\_Manager'$	$\{(t, r) \mid t \bmod 42 \in [37, 41)\}$
$Accepting\_Mode\_Manager'$	$\{(t, r) \mid t \bmod 21 \in [0, 9)\}$

It is easily seen that the above resource sets are disjoint, so by lemma 2.2 it is enough to design a scheduler ϕ such that:

$$
\begin{aligned}
&\mathcal{R}[\![\mathit{Display\_Manager}' \mid \phi]\!] \Rightarrow \\
&\qquad R_{req} \subseteq \{(t,r) \mid t \bmod \mathit{21} \in [\mathit{9},\mathit{16})\} \wedge \neg\, \mathsf{fail}, \\
&\mathcal{R}[\![\mathit{Confirmation\_Light\_Manager}' \mid \phi]\!] \Rightarrow \\
&\qquad R_{req} \subseteq \{(t,r) \mid t \bmod \mathit{42} \in [\mathit{16},\mathit{21})\} \wedge \neg\, \mathsf{fail}, \\
&\mathcal{R}[\![\mathit{Screen\_Clear\_Manager}' \mid \phi]\!] \Rightarrow \\
&\qquad R_{req} \subseteq \{(t,r) \mid t \bmod \mathit{42} \in [\mathit{37},\mathit{41})\} \wedge \neg\, \mathsf{fail}, \\
&\mathcal{R}[\![\mathit{Accepting\_Mode\_Manager}' \mid \phi]\!] \Rightarrow \\
&\qquad R_{req} \subseteq \{(t,r) \mid t \bmod \mathit{21} \in [\mathit{0},\mathit{9})\} \wedge \neg\, \mathsf{fail}.
\end{aligned}
$$

We examine the first of these in detail; the rest are similar. We label the display manager to obtain:

$$
\begin{aligned}
&\mathsf{var}\ \mathit{current\_kybd}, \mathit{current\_acc}, \mathit{current\_ts}, \mathit{previous\_ts}\ \mathsf{in} \\
&[\mathit{21}] \begin{pmatrix} L_{i1} :: \mathit{current\_ts} := \mathit{Now}; \\ L_{i2} :: \mathit{current\_kybd} \leftarrow \mathit{KYBD}; \\ L_{i3} :: \mathit{current\_acc} \leftarrow \mathit{ACC}; \\ L_{i4} :: \mathit{current\_ts} := \max\{\mathit{current\_ts}, \mathit{current\_kybd.ts}, \mathit{current\_acc.ts}\} \end{pmatrix}; \\
&\mathsf{loop\ for}\ n\ \mathsf{period}\ \mathit{21} \\
&\qquad L_{l1} :: \mathit{previous\_ts} := \mathit{current\_ts}; \\
&\qquad L_{l2} :: \mathit{current\_ts} := \mathit{Now}; \\
&\qquad L_{l3} :: \mathit{current\_kybd} \leftarrow \mathit{KYBD}; \\
&\qquad L_{l4} :: \mathit{current\_acc} \leftarrow \mathit{ACC}; \\
&\qquad L_{l5} :: \mathit{current\_ts} := \max\{\mathit{current\_ts}, \mathit{current\_kybd.ts}, \mathit{current\_acc.ts}\}; \\
&\qquad L_{l6} :: \mathsf{if}\quad \mathit{current\_kybd.ts} > \mathit{previous\_ts} \wedge \mathit{current\_acc.v} \Rightarrow \\
&\qquad\qquad\qquad L_{l7} :: \mathit{current\_kybd.v} \rightarrow \mathit{DSPLY} \\
&\qquad\qquad \mathsf{fi} \\
&\mathsf{endloop}.
\end{aligned}
$$

And we define a scheduler such that the commands labelled L_{i1} and L_{l1} are delayed by 9 seconds—so that they start at time $\mathit{21}k + \mathit{9}$ for some k—and schedule each of the other commands immediately:

$$
\begin{array}{lcllcl}
\phi(t, L_{i1}) & \widehat{=} & \{(t+\mathit{9}, r)\}, & \phi(t, L_{i2}) & \widehat{=} & \{(t, r)\}, \\
\phi(t, L_{i3}) & \widehat{=} & \{(t, r)\}, & \phi(t, L_{i4}) & \widehat{=} & \{(t, r)\}, \\
\phi(t, L_{l1}) & \widehat{=} & \{(t+\mathit{9}, r)\}, & \phi(t, L_{l2}) & \widehat{=} & \{(t, r)\}, \\
\phi(t, L_{l3}) & \widehat{=} & \{(t, r)\}, & \phi(t, L_{l4}) & \widehat{=} & \{(t, r)\}, \\
\phi(t, L_{l5}) & \widehat{=} & \{(t, r)\}, & \phi(t, L_{l6}) & \widehat{=} & \{(t, r)\}, \\
\phi(t, L_{l7}) & \widehat{=} & \{(t, r)\}. & & &
\end{array}
$$

It is easily verified that the program uses only the available resources, and does not fail because the deadline is met.

The scheduler we have defined fits into the *deadline monotonic* class of schedulers [ABRW91].

Finally, we collect together all the time constraints (taking the minimum possible values) to obtain:

$$
\begin{array}{rclrclrcl}
t'_{dsply} & = & \mathit{41}, & t'_{conf} & = & \mathit{83}, & t'_{sclr} & = & \mathit{83}, \\
t_{acc} & = & \mathit{41}, & t_{dsply} & = & \mathit{41}, & t_{conf} & = & \mathit{124}, \\
t_{sclr} & = & \mathit{124}, & t_{kmin} & = & \mathit{83}, & t_{init} & = & \mathit{42}.
\end{array}
$$

5 Conclusions

In this paper we have presented a case study using the Temporal Agent Model. We have described a method of separating the functional and temporal parts of a specification using a *delayed specification*; we believe that this method makes formalizing many requirements much easier and less error prone. We have presented a theorem (theorem 3.4) that allows us to chain two such delayed specifications together: this is useful when we introduce new intermediate shunts which themselves satisfy a delayed specification. We have also presented a theorem (theorem 4.1) that, under certain circumstances, allows us to refine such a delayed specification into code. We have used delayed specifications to capture our system requirements, and then used the theorems to derive a correct implementation.

We believe that the TAM refinement calculus, particularly using this method of delayed specifications, provides a very useful tool in developing real-time systems. We would like to undertake further related case studies, with the hope of extending the underlying theory, producing more useful theorems, and making the existing theorems easier to use. For example, the hardest part of applying theorem 3.4 is proving condition 2; it would be useful to develop lemmas showing when this condition is true.

The observant reader will have noticed that the system we have derived satisfies tighter deadlines than we have claimed for it. For example, the component $\mathit{Display\_Manager}'$ (with the given scheduler) satisfies TR'_1 with a value for t_{dsply} of 24, whereas we have only claimed this with a value of 41. The reason for the discrepancy is that when we have a scheduler there is much more structure to the design: for example in the case of the display manager, the shunt reads always occur at times $21.k + 11$ and $21.k + 12$, and the write occurs at time $21.k + 15$ ($k \in 1 \mathinner{..} n$). Repeating the proof of theorem 4.1 including consideration of the scheduler would generate these smaller time parameters.

The method of delayed specifications that we have introduced in this paper are applicable not only to TAM: it would be interesting to adapt them to other formalisms, such as Timed CSP.

One weakness with the current TAM model is that it can deal only with *finite* agents, i.e. agents that must eventually terminate; this runs counter to the paradigm of reactive systems, which may execute for ever. In theory we can get round this by specifying that a system must operate correctly up until time T, and then taking T arbitrarily large; but in practice this is awkward for we must consider the value of T in our proofs. It would therefore be useful to extend the semantic models of TAM to include infinite behaviour. This question is considered in [Low95].

References

[ABRW91] N. C. Audsely, A. Burns, M. F. Richardson, and A. L. Wellings. Hard real-time scheduling: The deadline monotonic approach. In *Proc. 8th IEEE Workshop on Real-Time Operating Systems and Software*, pages 15–17, 1991.

[AK86] B. Auernheimer and R. Kemmerer. RT-ASLAN: A specification language for real-time systems. *IEEE Transactions on Software Engineering*, 12, 1986.

[Bac88] R. J. R. Back. A calculus of refinements for program derivations. *Acta Informatica*, 25:593–624, 1988.

[Bac89] R. J. R. Back. Refinement calculus, part II: Parallel and reactive programs. In de Baker, de Roever, and Rozenberg, editors, *Stepwise Re-*

finement of Distributed Systems, volume 430 of *LNCS*. Springer-Verlag, 1989.

[BW90] R. J. R. Back and J. Wright. Refinement concepts formalised in HOL. *Formal Aspects of Computing*, 3(4), 1990.

[Dav93] Jim Davies. *Specification and Proof in Real-Time Systems.* Cambridge University Press, 1993. D. Phil thesis, Oxford University, 1991.

[He91] Jifeng He. Specification oriented semantics for ProCoS programming language PL^{time}, 1991. PRG-OU-HJF-71.

[HL87] L. Heitmeyer and T. Labaw. The A-7E aircraft in RTL. Technical Report 25, Washington University, 1987.

[JM86] F. Jahanian and A. K. Mok. Safety analysis of timing properties in real-time system. *IEEE Transactions on Software Engineering*, 12, 1986.

[Low94] Gavin Lowe. Scheduling-oriented models for real-time systems. Submitted for publication; also available as Technical Report PRG-TR-10-94, Oxford University Computing Laboratory, 1994.

[Low95] Gavin Lowe. Infinite behaviours in the temporal agent model. Technical Report PRG-TR-3-95, Oxford University Computing Laboratory, 1995.

[LZ95] Gavin Lowe and Hussein Zedan. Refinement of complex systems: A case study. Technical Report PRG-TR-2-95, Oxford University Computing Laboratory, 1995.

[McD89] J. McDermid. *The theory and Practice of Refinement.* Butterworths, 1989.

[MH92] B. P. Mahoney and I. J. Hayes. A case-study in timed refinement: A mine pump. *IEEE Transactions on Software Engineering*, 18(9):817–826, 1992.

[Mor90] Carroll Morgan. *Programming from Specifications.* Prentice Hall International, 1990.

[Sch93] D. Scholefield. TAM technical notes 1 and 2. Technical Report YCS 213, University of York, 1993.

[SZH94] D. Scholefield, H. Zedan, and J. He. A specification oriented semantics for the refinement of real-time systems. *Theoretical Computer Science*, 130, August 1994.

A Some refinement laws

In this appendix we list the refinement laws that we have used in the case study.

Law A.1 (Strengthen specification): If $\Phi' \Rightarrow \Phi$ then $w : \Phi \sqsubseteq w : \Phi'$.

Law A.2 (Contract frame): $w \cup x : \Phi \wedge \mathsf{stable}(x, t_\alpha, t_\omega) \sqsubseteq w : \Phi$.

Law A.3 (New shunt): $w : \Phi \sqsubseteq \mathsf{shunt}\, w' \;\mathsf{in}\; w \cup w' : \Phi$.

Law A.4 (New variable): $w : \Phi \sqsubseteq \mathsf{var}\ w'\ \mathsf{in}\ w \cup w' : \Phi$.

Law A.5 (Parallel composition): If $w \cap w' = \emptyset$, and t_ω does not appear free in Φ or Φ', then

$$w \cup w' : \Phi \wedge \Phi' \ \sqsubseteq\ (w : \Phi) \mid (w' : \Phi').$$

Law A.6 (Sequential composition):

$$w : \Big(\exists\, t \bullet \Phi[t/t_\omega] \wedge \Phi'[t/t_\alpha]\Big) \ \sqsubseteq\ (w : \Phi)\ ;\ (w : \Phi').$$

Law A.7 (Periodic loop): For $p \in \mathbb{Z}^+$ (the period of the loop):

$$w : \Big(\forall\, k \in [0, n) \bullet \Phi[p.k/t_\alpha, (p+1).k/t_\omega]\Big) \ \sqsubseteq\ \mathsf{loop\,for}\ n\ \mathsf{period}\ p\ \ (w : \Phi).$$

Law A.8 (Alternation): If

$$\begin{pmatrix} \exists\, t > 0\ ;\ i \in I \bullet g_i \wedge \Phi_i[t_\alpha + t/t_\alpha] \wedge \mathsf{stable}(w, t_\alpha, t_\alpha + t) \\ \vee \\ \forall\, i \in I \bullet \neg\, g_i \wedge \mathsf{stable}(w, t_\alpha, t_\omega) \wedge t_\omega > t_\alpha \end{pmatrix} \ \Rightarrow\ \Phi$$

then

$$w : \Phi \ \sqsubseteq\ \mathsf{if}\ \mathop{\Box}_{i \in I} g_i \Rightarrow w : \Phi_i\ \ \mathsf{fi}.$$

Law A.9 (Deadline): $w : (\Phi \wedge t_\omega - t_\alpha = t) \ \sqsubseteq\ [t]\ (w : \Phi)$.

Law A.10 (Shunt write): If $s \in w$ then $w : \mathsf{write}(s, x, [t_\alpha, t_\omega]) \sqsubseteq x \to s$.

Law A.11 (Shunt read): If $x \in w$ then

$$w : \Big(\exists\, t \in [t_\alpha, t_\omega] \bullet x@t_\omega = s@t\Big) \ \sqsubseteq x \leftarrow s.$$

Law A.12 (Assignment): If $x \in w$ then

$$w : \Big(\exists\, t \in [t_\alpha, t_\omega] \bullet x@t_\omega = e@t\Big) \ \sqsubseteq x := e.$$