Scheduling-Oriented Models for Real-Time Systems

Gavin Lowe*

July 10, 1995

Abstract

In this paper, we define a formal model for reasoning about resource allocation and scheduling in real-time systems. We extend the model in [SZJ94], which models the functionality of agents in the language TAM. Our extended model allows us to argue about the resource requirements of real-time distributed systems, and so identify conflicts upon resources. We can use the model to define and reason about schedulers that map jobs onto resources. The model will aid in the important transformation from an initial design for a system to an actual implementation with jobs scheduled on processors.

1 Introduction

The performance of a real-time system depends upon the resources, such as CPUs and memory, available. For example, if a program has two parallel threads, but only one CPU is available, then we need to find a way of allocating the resource between the two threads, or—equivalently—scheduling the threads on the CPU. This paper develops a method of formally reasoning about resource allocation and scheduling.

One approach to this question has been to adopt the *unbounded parallelism hypothesis*, i.e., the assumption that sufficient resources are always available. However, implementations of real-time systems rarely achieve unbounded parallelism, because fluctuations in work load mean that having sufficient resources at peak times would result in considerable spare capacity at low load times, reducing cost effectiveness.

In this paper we develop semantic models that allow us to argue about resource allocation and scheduling. The models are based upon the language TAM [SZJ94, SZ92, SZ93], a wide-spectrum development language for real-time systems, with constructs for such features as communication, deadline and parallelism.

It is possible to reason about resource allocation in event-based process algebras, such as Timed CSP [RR86, DS95, DRRS95] or timed versions of CCS [MT90, Yi90, HR90], by modelling resources as processes. This approach can work well when the program itself is easily modelled as a process; however, event-based process algebras tend to have a very different syntax to most traditional programming languages, so this modelling can be somewhat unnatural. In such cases, it seems more natural to consider a formalism, such as TAM, that is closer to a traditional programming language.

Further, using the process algebra approach seems to make it difficult to reason about a system *compositionally*: the approach forces us to model resources globally rather than locally. We prefer to model resources in a more abstract manner: our approach is to calculate the *amount* of resources required by each component separately, and then combine these to decide how to allocate resources between components, working at a system level; finally we schedule each component on the resources allocated to it, reasoning locally.

Our models will enable us to define and reason about schedulers which allocate resources to tasks. They will recognize that if two agents both try to access the same resource simultaneously, then the system will fail. The models will also tell us whether the resources available are sufficient to ensure that all deadlines are met.

In [SZ93] a refinement calculus is given for TAM. This allows an abstract specification to be refined into an initial design via a number of provably correct transformations. We take this one step further: we transform the initial design into an implementation with components scheduled on resources in a correct manner.

We believe that issues of scheduling should be dealt with *after* the initial design, because if the available

*Author's address: Oxford University Computing Laboratory, Wolfson Building, Parks Road, Oxford, OX1 3QD, United Kingdom; tel. +1865 283603; fax +1865 273839; e-mail: gavin.lowe@comlab.ox.ac.uk

resources change, we do not want to have to redo too much of the derivation: the functionality-oriented refinement to the initial design should remain the same; only the scheduling calculations will change. Also, program derivation can be seen as a top-down activity, based upon splitting a specification into specifications for subcomponents, whereas scheduling calculations are more bottom-up, considering first the resource requirements of atomic commands, and then combining these to derive the resource requirements of the whole program; this disparity suggests that it is difficult to perform the two activities together. Thus the refinement scheme is as illustrated by Figure 1.

$$
\begin{array}{c}
\text{Specification} \\
\Big\downarrow \ \text{Functionality-oriented refinement calculus} \\
\text{Initial design} \\
\Big\downarrow \ \text{Resource-oriented refinement calculus} \\
\text{Implementation}
\end{array}
$$

Figure 1: The two stage implementation process

Gerber and Lee [GL89, GL90, GL92] have produced a language, the *Calculus for Communicating Shared Resources*, for reasoning about resource requirements in real-time systems. Their model is operational, and similar to that of SCCS [Mil89], extended to model priorities (following the approach of [CH88]), and such that transitions are allowed only if they do not involve a conflict on resources. Their approach differs from ours in that we use a compositional denotational model, which allows us to break down requirements on systems into requirements on the subcomponents. Also, we define a hierarchy of models, allowing us to separate the issues of functionality and scheduling.

In the next section, we describe the language TAM, and present the semantics given in [SZJ94]: this represents an agent by a predicate describing its behaviour. We illustrate the model with an example in Section 3.

In Section 4, we extend the model so as to be able to discuss the resources available to, and used by, an agent; resources are allocated to jobs according to a fixed (i.e. compile-time) scheduler; we also model the fact that an agent may fail, either because insufficient resources are available, or because it fails to meet a deadline; we illustrate the model by applying it to the running example.

We extend the model in Section 5, to be able to discuss dynamic schedulers, where scheduling decisions may be based upon the checkpoints encountered so far in the execution. Thus we will define a hierarchy of models, as in Figure 2. We discuss a number of possible extensions to the models in Section 6.

$$
\begin{array}{c}
\text{Dynamic-scheduling model (Section 5)} \\
\Big| \\
\text{Fixed-scheduling model (Section 4)} \\
\Big| \\
\text{Functional model (Section 2)}
\end{array}
$$

Figure 2: The hierarchy of models.

2 A functional semantics for TAM

The Temporal Agent Model (TAM) is a wide spectrum language for modelling real-time distributed computer systems. A TAM system is a collection of parallel agents, each with its own set of variables. Agents communicate via a form of asynchronous channel called a *shunt*. A shunt holds a pair consisting of a time-stamp, giving the last time at which the shunt was updated, and a value. Each shunt is owned by a single agent that can write values to it; all agents can then read this value.

We consider the subset of the TAM language defined by the BNF expression:

$$
\begin{array}{rcl}
\mathcal{A} & ::= & w : \Phi \;\big|\; \mathsf{skip} \;\big|\; \Delta t \;\big|\; x := E \;\big|\; (x, y) \leftarrow s \;\big| \\
& & E \rightarrow s \;\big|\; [t]\mathcal{A} \;\big|\; \mathcal{A} \,;\, \mathcal{A} \;\big|\; \mathcal{A} \sqcap \mathcal{A}' \;\big| \\
& & \bigsqcup_{t\, i \in I} g_i \Rightarrow \mathcal{A}_i \;\big|\; \mathcal{A} \mid \mathcal{A} \;\big|\; \mu_n \, \mathcal{A} \,,
\end{array}
$$

where x and y are variables, s is a shunt, w is a set of variables and shunts, Φ is a specification predicate, t is a time (which we take to be a natural number), E is an expression on variables, I is a finite indexing set, n is a natural number, and g_i is a boolean expression.

In [SZJ94] a semantics was given for this language, representing an agent by a predicate upon the start and termination times, and the values held in shunts and variables. Below we give informal descriptions of each of the language constructs, and give the predicate semantics. Note that our semantics differs a little from the semantics of [SZJ94]: we have made a few minor changes so as to suit our needs.

An agent is represented by a predicate Φ in the *TAM Logic Language* (*TAMLL*): a first-order logic with simple extensions to reason about time and the values held in variables and shunts. The predicate Φ may contain two distinguished free variables, t_α and t_ω, representing the start and termination times of an execution. We also allow terms of the form $s@t$ to refer to the value held in shunt s at time t, and similarly $x@t$ to refer to the value held in variable x. We write $s@t.ts$ for the time-stamp of s at t, and $s@t.v$ for the value.

For example, the predicate

$$t_\omega \leq t_\alpha + 6 \wedge \exists\, t \in [t_\alpha, t_\omega) \bullet out@t_\omega.v = (in@t.v)!$$

says that the agent terminates within 6 time units, and the final value in the *out* shunt is the factorial of the value found in the *in* shunt at some time t during the execution. Note that as shunts are *asynchronous* channels, more than one write to *in* may occur, but the above predicate only specifies that one input must be processed.

Note that this model does not consider the possibility of failures, for example when an agent fails to meet a deadline. Later, when we consider resource considerations and scheduling, we will also model the possibility of failure: we believe that it is possible to decide whether a deadline is met only when we know what resources are available to the agent.

All shunts obey the following law:

Law 2.1: $\forall\, t : Time \bullet s@t.ts \leq t \wedge (s@t.ts < t \Rightarrow s@t = s@(t-1))$.

The time-stamps of shunts represent the last time at which they were written to, so: (1) they will always be no more than the current time; and (2) if the time-stamp is strictly less than the current time, then the shunt has not been written to at this time step, and so its value must be the same as at the previous time step.

We will define a semantic function

$$\mathcal{F} : TAM \to TAMLL,$$

such that $\mathcal{F}[\![\mathcal{A}]\!]$ gives the predicate describing the functionality of agent $\mathcal{A}$.

We will need a predicate that asserts that the value of a variable or shunt does not change during an execution:

$$stable(s, t_\alpha, t_\omega) \mathrel{\widehat=} \forall\, t \in [t_\alpha, t_\omega] \bullet s@t = s@t_\alpha.$$

This definition extends to sets of variables and shunts in the obvious way.

The agent $w : \Phi$ is a specification statement which says that only variables and shunts in the frame w may be changed, and the execution must satisfy the predicate Φ:

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

Here and subsequently we take W to be the alphabet of the agent in question. The refinement rules of [SZJ94, SZ92] allow specifications of this form to be refined into an initial design in "concrete TAM".

The agent skip terminates without changing the values of any variables or shunts:

$$\mathcal{F}[\![\mathsf{skip}]\!] \mathrel{\widehat=} t_\omega \geq t_\alpha \wedge stable(W, t_\alpha, t_\omega).$$

The agent Δt terminates after t time units without changing the values of any variables or shunts:

$$\mathcal{F}[\![\Delta t]\!] \mathrel{\widehat=} t_\omega = t_\alpha + t \wedge stable(W, t_\alpha, t_\omega).$$

The agent $x := E$ evaluates the expression E using the *initial* values of variables, and assigns this value to x:

$$\begin{aligned}\mathcal{F}[\![x := E]\!] \mathrel{\widehat=}\ & t_\omega > t_\alpha \wedge stable(W \setminus x, t_\alpha, t_\omega)\\ & \wedge x@t_\omega = E@t_\alpha.\end{aligned}$$

We write $E@t$ to represent the value of expression E where all the variables are taken at time t.

The agent $(x, y) \leftarrow s$ reads shunt s and stores the time-stamp in x and the value in y; the shunt read occurs at some time t during the execution:

$$\begin{aligned}\mathcal{F}[\![(x, y) \leftarrow s]\!] \mathrel{\widehat=}\ & t_\omega > t_\alpha \wedge stable(W \setminus \{x, y\}, t_\alpha, t_\omega)\\ & \wedge \exists\, t \in [t_\alpha, t_\omega) \bullet x@t_\omega = s@t.ts\\ & \qquad \wedge y@t_\omega = s@t.v.\end{aligned}$$

The agent $E \to s$ writes the value of expression E into shunt s, time-stamping it at the time of the write. We specify that agents cannot write to shunts at their start times.

$$\begin{aligned}\mathcal{F}[\![E \to s]\!] \mathrel{\widehat=}\ & t_\omega > t_\alpha \wedge stable(W \setminus s, t_\alpha, t_\omega)\\ & \wedge \exists\, t \in (t_\alpha, t_\omega] \bullet stable(s, t_\alpha, t-1)\\ & \qquad \wedge s@t = (t, E@t_\alpha)\\ & \qquad \wedge stable(s, t, t_\omega).\end{aligned}$$

The agent $[t]\mathcal{A}$ executes $\mathcal{A}$ but gives it a deadline of t: we are interested only in those executions of $\mathcal{A}$ that terminate within t time units; if $\mathcal{A}$ terminates before t time units are complete, then it idles for the rest of the period:

$$\begin{aligned}\mathcal{F}[\![[t]\mathcal{A}]\!] \mathrel{\widehat=}\ & \exists\, t'_\omega \bullet \mathcal{F}[\![\mathcal{A}]\!][t'_\omega/t_\omega] \wedge t'_\omega \leq t_\alpha + t = t_\omega\\ & \wedge stable(W, t'_\omega, t_\omega).\end{aligned}$$

$\mathcal{A} ; \mathcal{A}'$ forms a sequential composition of $\mathcal{A}$ and $\mathcal{A}'$:

$$\mathcal{F}[\![\mathcal{A} ; \mathcal{A}']\!] \mathrel{\widehat=} \exists\, t \in [t_\alpha, t_\omega] \bullet \mathcal{F}[\![\mathcal{A}]\!][t/t_\omega] \wedge \mathcal{F}[\![\mathcal{A}']\!][t/t_\alpha].$$

The agent $\mathcal{A} \sqcap \mathcal{A}'$ nondeterministically acts like $\mathcal{A}$ or $\mathcal{A}'$:

$$\mathcal{F}[\![\mathcal{A} \sqcap \mathcal{A}']\!] \mathrel{\widehat=} \mathcal{F}[\![\mathcal{A}]\!] \vee \mathcal{F}[\![\mathcal{A}']\!].$$

The agent $\bigsqcup_{i \in I}{}_t\, g_i \Rightarrow \mathcal{A}_i$ evaluates each of the guards g_i for $i \in I$, using the initial values of variables and shunts;

it then executes one of the $\mathcal{A}_i$ such that the corresponding g_i is *true*; if all the g_is evaluate to *false* then the agent simply terminates. It is assumed that the guards can be evaluated within time t, and that if necessary the agent idles to pad-out this interval.

$$
\begin{array}{l}
\mathcal{F}[\![\bigsqcup_{t \atop i\in I} g_i \Rightarrow \mathcal{A}_i]\!] \mathrel{\widehat{=}} \\
\quad \begin{array}{ll}
\text{let} & t'_\alpha = t_\alpha + t \;\text{ and }\; J = \{i \in I \mid g_i @ t_\alpha\} \;\text{ in} \\
\text{if} & J = \emptyset \;\text{ then }\; t_\omega = t'_\alpha \wedge \mathit{stable}(W, t_\alpha, t_\omega) \\
\text{else} & \mathit{stable}(W, t_\alpha, t'_\alpha) \wedge \exists\, j \in J \bullet \mathcal{F}[\![\mathcal{A}_j]\!][t'_\alpha / t_\alpha].
\end{array}
\end{array}
$$

$\mathcal{A} \mid \mathcal{A}'$ executes $\mathcal{A}$ and $\mathcal{A}'$ in parallel; we assume that the composition terminates as soon as both agents have terminated:

$$
\begin{array}{l}
\mathcal{F}[\![\mathcal{A}_1 \mid \mathcal{A}_2]\!] \mathrel{\widehat{=}} \\
\quad \exists\, t_1, t_2 \bullet \mathcal{F}[\![\mathcal{A}_1]\!][t_1 / t_\omega] \wedge \mathit{stable}(W_1, t_1, t_\omega) \\
\qquad\qquad \wedge\, \mathcal{F}[\![\mathcal{A}_2]\!][t_2 / t_\omega] \wedge \mathit{stable}(W_2, t_2, t_\omega) \\
\qquad\qquad \wedge\, t_\omega = t_1 \sqcup t_2,
\end{array}
$$

where W_1 and W_2 are the alphabets of $\mathcal{A}_1$ and $\mathcal{A}_2$, respectively. Once one component terminates, its variables remain stable until the other terminates.

The agent $\mu_n \bullet \mathcal{A}$ represents a *finite* iterative process, executing $\mathcal{A}$ n times:

$$
\begin{array}{rcl}
\mu_0\, \mathcal{A} & \mathrel{\widehat{=}} & \mathsf{skip}, \\
\mu_{n+1}\, \mathcal{A} & \mathrel{\widehat{=}} & \mathcal{A} \,;\, \mu_n\, \mathcal{A}.
\end{array}
$$

We can use this to define a periodically iterative process:

$$\mathsf{loop\ for}\ n\ \mathsf{period}\ p \bullet \mathcal{A} \;\mathrel{\widehat{=}}\; \mu_n\, [p]\mathcal{A}.$$

This process executes $\mathcal{A}$ n times with a period of p.

We can define a refinement relation between TAM agents as follows:

Definition 2.2: $\mathcal{A} \sqsubseteq_{\mathcal{F}} \mathcal{A}'$ *iff* $\mathcal{F}[\![\mathcal{A}]\!] \Leftarrow \mathcal{F}[\![\mathcal{A}']\!]$.

$\mathcal{A}$ is refined by $\mathcal{A}'$ if the behaviour of $\mathcal{A}'$ is more fully defined than that of $\mathcal{A}$. We will write $\sqsubseteq$ for this relation where this will not cause confusion. A number of refinement rules can be proved from the semantic definitions; the rules we will use in the rest of this paper are given in appendix A.

3 Example

We consider an example, based upon [Jac90], which aims to model part of the start-up routine for an aircraft engine. The system under consideration is shown in Figure 3. We aim to specify and develop a controller which sits between the pilot and the engine. The pilot may request air or fuel, or request for these to be turned off.

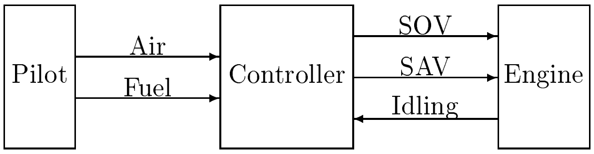

Figure 3: The simplified aircraft engine start-up system

The controller may send signals to the engine to open or close two valves: the fuel shut-off valve (SOV) and the starter air valve (SAV). The engine can return signals to the controller to inform it that it has reached idle conditions.

The controller should open and close the valves according to certain conditions, taken from [Jac90]. Basically, the valves are opened and closed as requested by the pilot, but some safety checks are made to prevent dangerous actions by the pilot. We split the specification into liveness and safety conditions. The liveness conditions are:

L_1: If the pilot requests for the fuel to be turned off, and the shut-off valve is open, then a signal should be sent for the shut-off valve to be closed;

L_2: The shut-off valve should be opened if the pilot requires fuel, as long as the starter air valve is already open;

L_3: The starter air valve should be closed when the pilot requests the air to be turned off;

L_4: Once the engine reaches idling conditions, the starter air valve should be closed automatically;

L_5: If the pilot requests air, and the engine has not yet reached idling conditions, then the starter air valve should be opened.

The safety conditions are that the valves may change state *only* as described by the liveness conditions:

S_1: The shut-off valve may be closed only if the pilot asks for the fuel to be turned off;

S_2: The shut-off valve may be opened only if the pilot asks for the fuel to be turned on and the starter air valve is open;

S_3: The starter air valve may be closed only if the pilot asks for the air to be turned off, or the engine is idling;

S_4: The starter air valve may be opened only if the pilot requests air, and the engine is not idling.

We will formalize these conditions below.

In an ideal system, the valves should be opened or closed immediately, but in a realistic implementation there is bound to be some delay involved in sampling the input shunts and reacting accordingly. We will make an assumption that the values held in input shunts do not change too quickly, so new inputs will not be missed by periodic sampling. Our liveness specifications will therefore be of the form: if some condition holds of the input channels for some period of time, then one of the valves will be opened or closed within some interval. Our safety specifications will be of the form: if some condition holds of the valves, then the input shunts have recently held corresponding (safe) values.

3.1 Formal specification

As explained above, we need to assume that the values held in input shunts do not change too quickly. In order to formalize this, we introduce a couple of specification macros. We write $\mathsf{write}(s,k,t)$ to specify that value k is written to s at time t; we generalize this to specify that a shunt is written to during some interval:

$$\begin{aligned}\mathsf{write}(s,k,t) &\mathrel{\widehat=} s@t=(t,k),\\ \mathsf{write}(s,k,t,t') &\mathrel{\widehat=} \exists\, t''\in[t,t']\bullet \mathsf{write}(s,k,t'').\end{aligned}$$

The predicate $\mathsf{stable}(s,k,t,t')$ is true if the shunt s holds value k throughout the interval $[t,t']$:

$$\mathsf{stable}(s,k,t,t') \mathrel{\widehat=} \forall\, t''\in[t,t']\bullet s@t''.v=k.$$

We can now formalize our assumption about inputs not changing too quickly. We specify that if a value is written to an input shunt at time t, then that shunt remains stable for the following t_{stab} time units:

$$\begin{aligned}\mathit{ASS} \mathrel{\widehat=} \forall\, s\in\{\mathit{fuel},\mathit{air},\mathit{idling}\}\bullet \forall\, t\in \mathit{Time}\bullet\\ \mathsf{write}(s,k,t)\Rightarrow \mathsf{stable}(s,k,t,t+t_{stab}).\end{aligned}$$

The customer must agree to meet this assumption in order to be sure that our program meets his requirements. Here t_{stab} is a formal parameter; when we derive our system, we will calculate a value for t_{stab}, and the customer must decide whether this value is acceptable.

When we formalize the requirements, below, we will specify that the appropriate valves are opened or closed within t_{del} time units. Again, we will calculate a value for t_{del} when we derive our system, and the customer must decide whether this value is acceptable. The reader may safely assume $t_{del}>t_{stab}$.

We will insist that the requirements continue to be met until some arbitrarily large time, which we will write as T.

We now formalize our requirements. If the pilot requests for the fuel to be turned off, and the shut-off valve is open, then a signal should be sent for the shut-off valve to be closed:

$$\begin{aligned}L_1 \mathrel{\widehat=} \forall\, t\in[t_\alpha,T]\bullet\\ \mathit{SOV}@t.v=\mathit{open}\wedge \mathsf{write}(\mathit{fuel},\mathit{off},t)\Rightarrow\\ \mathsf{write}(\mathit{SOV},\mathit{close},t,t+t_{del}).\end{aligned}$$

In fact our specification says that if the pilot requests for the fuel to be turned off, then the shut-off valve will be closed within t_{del} time units. The assumption ASS tells us that if he asks for fuel at time t, he will not change his mind within t_{stab} time units.

The shut-off valve should be opened if the pilot requires fuel, as long as the starter air valve is open:

$$\begin{aligned}L_2 \mathrel{\widehat=} \forall\, t\in[t_\alpha,T]\bullet\\ \begin{pmatrix}\mathit{SOV}@t.v=\mathit{closed}\\ \wedge\ \mathsf{stable}(\mathit{SAV},\mathit{open},t,t+t_{stab})\\ \wedge\ \mathsf{write}(\mathit{fuel},\mathit{on},t)\end{pmatrix}\Rightarrow\\ \mathsf{write}(\mathit{SOV},\mathit{open},t,t+t_{del}).\end{aligned}$$

The starter air valve should be closed when the pilot requests the air to be turned off:

$$\begin{aligned}L_3 \mathrel{\widehat=} \forall\, t\in[t_\alpha,T]\bullet\\ \mathit{SAV}@t.v=\mathit{open}\wedge \mathsf{write}(\mathit{air},\mathit{off},t)\Rightarrow\\ \mathsf{write}(\mathit{SAV},\mathit{close},t,t+t_{del}).\end{aligned}$$

Once the engine reaches idling conditions, the starter air valve should be closed:

$$\begin{aligned}L_4 \mathrel{\widehat=} \forall\, t\in[t_\alpha,T]\bullet\\ \mathit{SAV}@t.v=\mathit{open}\wedge \mathit{idling}@t.v=\mathit{true}\Rightarrow\\ \mathsf{write}(\mathit{SAV},\mathit{close},t,t+t_{del}).\end{aligned}$$

If the pilot requests air and the engine has not yet reached idling conditions, then the starter air valve should be opened:

$$\begin{aligned}L_5 \mathrel{\widehat=} \forall\, t\in[t_\alpha,T]\bullet\\ \begin{pmatrix}\mathit{SAV}@t.v=\mathit{closed}\\ \wedge\ \mathsf{stable}(\mathit{idling},\mathit{false},t,t+t_{stab})\\ \wedge\ \mathsf{write}(\mathit{air},\mathit{on},t)\end{pmatrix}\Rightarrow\\ \mathsf{write}(\mathit{SAV},\mathit{open},t,t+t_{del}).\end{aligned}$$

Note that if the engine reaches idling condition just before the SAV is opened, during the interval $(t+t_{stab},t+t_{del})$, then the SAV will be closed again, by L_4.

The shut-off valve may be closed only if the pilot has recently asked for the fuel to be turned off:

$$\begin{aligned}S_1 \mathrel{\widehat=} \forall\, t\in[t_\alpha,T]\bullet\\ \mathsf{write}(\mathit{SOV},\mathit{close},t)\Rightarrow\\ \exists\, t'\in[t-t_{del},t]\bullet \mathit{fuel}@t'.v=\mathit{off}.\end{aligned}$$

The shut-off valve may be opened only if the pilot asks for the fuel to be turned on and the starter air valve is open:

$$S_2 \mathrel{\widehat{=}} \forall\, t \in [t_\alpha, T] \bullet \\ \quad \mathsf{write}(SOV, open, t) \Rightarrow \\ \quad\quad \exists\, t' \in [t - t_{del}, t] \bullet fuel@t'.v = on \\ \quad\quad\quad \wedge\, SAV@t'.v = open.$$

The starter air valve may be closed only if the pilot asks for the air to be turned off, or the engine is idling:

$$S_3 \mathrel{\widehat{=}} \forall\, t \in [t_\alpha, T] \bullet \\ \quad \mathsf{write}(SAV, close, t) \Rightarrow \\ \quad\quad \exists\, t' \in [t - t_{del}, t] \bullet air@t'.v = off \\ \quad\quad\quad \vee\, idling@t'.v = true.$$

The starter air valve may be opened only if the pilot requests air, and the engine is not idling:

$$S_4 \mathrel{\widehat{=}} \forall\, t \in [t_\alpha, T] \bullet \\ \quad \mathsf{write}(SAV, open, t) \Rightarrow \\ \quad\quad \exists\, t' \in [t - t_{del}, t] \bullet air@t'.v = on \\ \quad\quad\quad \wedge\, idling@t'.v = false.$$

Our complete specification is therefore:

$$\{SAV, SOV\} : ASS \Rightarrow \begin{pmatrix} L_1 \wedge L_2 \wedge L_3 \wedge L_4 \wedge L_5 \\ \wedge\, S_1 \wedge S_2 \wedge S_3 \wedge S_4 \end{pmatrix}.$$

The controller may write to the shunts SAV and SOV, and, provided the assumption holds, must ensure that L_1–L_5 and S_1–S_4 are satisfied.

3.2 Design

In this section we derive an agent that satisfies our specification; later we will show how this agent can be scheduled.

We use the refinement rule for parallel composition (Rule 2 in Appendix A) to split the specification into two parts, one dealing with the SOV, the other with the SAV; we refine the specification to:

$$\begin{array}{ll} & \{SOV\} : ASS \Rightarrow L_1 \wedge L_2 \wedge S_1 \wedge S_2 \\ | & \{SAV\} : ASS \Rightarrow L_3 \wedge L_4 \wedge L_5 \wedge S_3 \wedge S_4. \end{array}$$

We continue by considering the liveness conditions on the left hand side:

$$\begin{array}{ll} & ASS \Rightarrow L_1 \wedge L_2 \\ \Leftarrow & \langle \text{predicate calculus} \rangle \\ & \forall\, t \in [t_\alpha, T] \bullet \\ & \quad \begin{pmatrix} SOV@t.v = open \\ \wedge\, \mathsf{stable}(fuel, off, t, t + t_{stab}) \end{pmatrix} \Rightarrow \\ & \quad\quad \mathsf{write}(SOV, close, t, t + t_{del}) \\ & \quad \wedge \\ & \quad \begin{pmatrix} SOV@t.v = closed \\ \wedge\, \mathsf{stable}(SAV, open, t, t + t_{stab}) \\ \wedge\, \mathsf{stable}(fuel, on, t, t + t_{stab}) \end{pmatrix} \Rightarrow \\ & \quad\quad \mathsf{write}(SOV, open, t, t + t_{del}) \\ \Leftarrow & \langle \text{see below} \rangle \\ & \forall\, t \in [t_\alpha, T] \bullet \\ & \quad \begin{pmatrix} \mathsf{stable}(SOV, open, t, t + t_{stab}) \\ \wedge\, \mathsf{stable}(fuel, off, t, t + t_{stab}) \end{pmatrix} \Rightarrow \\ & \quad\quad \mathsf{write}(SOV, close, t, t + t_{del}) \\ & \quad \wedge \\ & \quad \begin{pmatrix} \mathsf{stable}(SOV, closed, t, t + t_{stab}) \\ \wedge\, \mathsf{stable}(SAV, open, t, t + t_{stab}) \\ \wedge\, \mathsf{stable}(fuel, on, t, t + t_{stab}) \end{pmatrix} \Rightarrow \\ & \quad\quad \mathsf{write}(SOV, open, t, t + t_{del}). \end{array}$$

To justify the last step we prove:

$$\begin{pmatrix} \begin{pmatrix} \mathsf{stable}(SOV, open, t, t + t_{stab}) \\ \wedge\, \mathsf{stable}(fuel, off, t, t + t_{stab}) \end{pmatrix} \Rightarrow \\ \mathsf{write}(SOV, close, t, t + t_{del}) \end{pmatrix} \Rightarrow \\ \begin{pmatrix} \begin{pmatrix} SOV@t.v = open \\ \wedge\, \mathsf{stable}(fuel, off, t, t + t_{stab}) \end{pmatrix} \Rightarrow \\ \mathsf{write}(SOV, close, t, t + t_{del}) \end{pmatrix}.$$

(The other condition is proven similarly.) Assume the antecedent and the left hand side of the consequent. Then either (1) SOV is stable during $(t, t + t_{stab}]$, in which case $\mathsf{write}(SOV, close, t, t + t_{del})$ from the antecedent, or (2) SOV is not stable during $(t, t + t_{stab}]$, in which case $\mathsf{write}(SOV, close, t, t + t_{stab})$, and so $\mathsf{write}(SOV, close, t, t + t_{del})$ because $t_{del} > t_{stab}$.

We aim to refine the specification to a periodic loop with period p, iterating n times. Choose n, p, t'_{stab} and t'_{del} such that $(n - 1).p \geq T - t_\alpha$, and

$$t'_{stab} + p - 1 \leq t_{stab}, \qquad t'_{del} + p - 1 \leq t_{del}.$$

Then taking $k = \left\lceil \frac{t - t_\alpha}{p} \right\rceil$ (so if $t \in [t_\alpha, T]$ then $k \in [0, n-1]$), we have $k.p \geq t - t_\alpha > (k-1).p$, so $t_\alpha + k.p \geq t$, and $t_\alpha + k.p + t'_{stab} \leq t + t_{stab}$, and $t_\alpha + k.p + t'_{del} \leq t + t_{del}$. Hence $[t_\alpha + k.p, t_\alpha + k.p + t'_{stab}] \subseteq [t, t + t_{stab}]$ and $[t_\alpha + k.p, t_\alpha + k.p + t'_{del}] \subseteq [t, t + t_{del}]$. Thus the above condition can be strengthened to:

$$\begin{array}{l} \forall\, k \in [0, n - 1] \bullet \\ \quad \begin{pmatrix} \mathsf{stable}(SOV, open, t_\alpha{+}k.p, t_\alpha{+}k.p{+}t'_{stab}) \\ \wedge\, \mathsf{stable}(fuel, off, t_\alpha{+}k.p, t_\alpha{+}k.p{+}t'_{stab}) \end{pmatrix} \\ \quad\quad \Rightarrow \mathsf{write}(SOV, close, t_\alpha{+}k.p, t_\alpha{+}k.p{+}t'_{del}) \\ \quad \wedge \\ \quad \begin{pmatrix} \mathsf{stable}(SOV, closed, t_\alpha{+}k.p, t_\alpha{+}k.p{+}t'_{stab}) \\ \wedge\, \mathsf{stable}(SAV, open, t_\alpha{+}k.p, t_\alpha{+}k.p{+}t'_{stab}) \\ \wedge\, \mathsf{stable}(fuel, on, t_\alpha{+}k.p, t_\alpha{+}k.p{+}t'_{stab}) \end{pmatrix} \\ \quad\quad \Rightarrow \mathsf{write}(SOV, open, t_\alpha{+}k.p, t_\alpha{+}k.p{+}t'_{del}). \end{array}$$

Hence:

$$\begin{array}{ll} & \{SOV\} : ASS \Rightarrow L_1 \wedge L_2 \wedge S_1 \wedge S_2 \\ \sqsubseteq & \left\langle \begin{array}{l} \text{strengthen specification (Rule 1),} \\ \text{using the above} \end{array} \right\rangle \end{array}$$

$$
\begin{array}{l}
\{SOV\} : \forall\, k \in [0, n-1] \bullet \\
\quad \left(\begin{array}{l}\mathsf{stable}(SOV, open, t_\alpha + k.p, t_\alpha + k.p + t'_{stab}) \\ \wedge\ \mathsf{stable}(fuel, off, t_\alpha + k.p, t_\alpha + k.p + t'_{stab})\end{array}\right) \\
\qquad \Rightarrow \mathsf{write}(SOV, close, t_\alpha + k.p, t_\alpha + k.p + t'_{del}) \\
\quad \wedge \\
\quad \left(\begin{array}{l}\mathsf{stable}(SOV, closed, t_\alpha + k.p, t_\alpha + k.p + t'_{stab}) \\ \wedge\ \mathsf{stable}(SAV, open, t_\alpha + k.p, t_\alpha + k.p + t'_{stab}) \\ \wedge\ \mathsf{stable}(fuel, on, t_\alpha + k.p, t_\alpha + k.p + t'_{stab})\end{array}\right) \\
\qquad \Rightarrow \mathsf{write}(SOV, open, t_\alpha + k.p, t_\alpha + k.p + t'_{del}) \\
\quad \wedge \\
\quad S_1 \wedge S_2 \\
\sqsubseteq\ \langle \text{introduce loop with period p (Rule 3)} \rangle \\
\mathsf{loop\ for}\ n\ \mathsf{period}\ p \bullet \\
\quad \{SOV\} : \\
\qquad \left(\begin{array}{l}\mathsf{stable}(SOV, open, t_\alpha, t_\alpha + t'_{stab}) \\ \wedge\ \mathsf{stable}(fuel, off, t_\alpha, t_\alpha + t'_{stab})\end{array}\right) \Rightarrow \\
\qquad\quad \mathsf{write}(SOV, close, t_\alpha, t'_{del}) \\
\qquad \wedge \\
\qquad \left(\begin{array}{l}\mathsf{stable}(SOV, closed, t_\alpha, t_\alpha + t'_{stab}) \\ \wedge\ \mathsf{stable}(SAV, open, t_\alpha, t_\alpha + t'_{stab}) \\ \wedge\ \mathsf{stable}(fuel, on, t_\alpha, t_\alpha + t'_{stab})\end{array}\right) \Rightarrow \\
\qquad\quad \mathsf{write}(SOV, open, t_\alpha, t'_{del}) \\
\qquad \wedge \\
\qquad \forall\, t \in [t_\alpha, t_\omega] \bullet \mathsf{write}(SOV, close, t) \Rightarrow \\
\qquad\quad \exists\, t' \in [t_\alpha, t] \bullet fuel@t'.v = off \\
\qquad \wedge \\
\qquad \forall\, t \in [t_\alpha, t_\omega] \bullet \mathsf{write}(SOV, open, t) \Rightarrow \\
\qquad\quad \exists\, t' \in [t_\alpha, t] \bullet fuel@t'.v = on \\
\qquad\qquad\qquad\qquad \wedge\ SAV@t'.v = open \\
\sqsubseteq\ \left\langle \begin{array}{l}\text{introduce alternation (Rule 4); strengthen} \\ \text{specification; take } t''_{del} = t'_{del} - t'_{stab} \text{ and} \\ \text{assume } t''_{del} > 0\end{array} \right\rangle \\
\mathsf{loop\ for}\ n\ \mathsf{period}\ p \bullet \\
\quad SOV.v = open \wedge fuel.v = off \Rightarrow \\
\qquad \{SOV\} : \left(\begin{array}{l}\mathsf{write}(SOV, close, t_\alpha, t_\alpha + t''_{del}) \\ \wedge\ \neg\ \mathsf{write}(SOV, open, t_\alpha, t_\omega)\end{array}\right) \\
\quad \bigsqcup_{t'_{stab}} \\
\quad \left(\begin{array}{l}SOV.v = closed \wedge SAV.v = open \\ \wedge\ fuel.v = on\end{array}\right) \Rightarrow \\
\qquad \{SOV\} : \left(\begin{array}{l}\mathsf{write}(SOV, open, t_\alpha, t_\alpha + t''_{del}) \\ \wedge\ \neg\ \mathsf{write}(SOV, close, t_\alpha, t_\omega)\end{array}\right) \\
\sqsubseteq\ \left\langle \begin{array}{l}\text{definition of shunt write (Rule 6),} \\ \text{deadline (Rule 5)}\end{array} \right\rangle \\
\mathsf{loop\ for}\ n\ \mathsf{period}\ p \bullet \\
\quad SOV.v = open \wedge fuel.v = off \Rightarrow \\
\qquad [t''_{del}]close \rightarrow SOV \\
\quad \bigsqcup_{t'_{stab}} \\
\quad \left(\begin{array}{l}SOV.v = closed \wedge SAV.v = open \\ \wedge\ fuel.v = on\end{array}\right) \Rightarrow \\
\qquad [t''_{del}]open \rightarrow SOV.
\end{array}
$$

In the above refinement we have assumed that the time parameters satisfy the following inequalities:

$$
\begin{array}{c}
t'_{stab} + p - 1 \leq t_{stab}, \\
t'_{del} + p - 1 \leq t_{del}, \\
(n-1).p \geq T - t_\alpha, \\
t''_{del} = t'_{del} - t'_{stab} > 0.
\end{array}
$$

A similar refinement process can be applied to the specification

$$\{SAV\} : ASS \Rightarrow L_3 \wedge L_4 \wedge L_5 \wedge S_3 \wedge S_4.$$

We obtain the program $P_1 \mid P_2$, where P_1 and P_2 are defined in Figure 4.

Later we will calculate a scheduler for this program, and so derive values for the time parameters.

4 A scheduling-oriented model

In this section, we extend the model of TAM so as to be able to argue about issues of resource allocation and scheduling.

We will represent the usage of resource r at time t by the pair (t, r). Formally, we define the space *Resource* of resources by

$$Resource \;\widehat{=}\; Time \times RName,$$

where *RName* is the space of resource names. We will use the term "resource" to refer to elements of either *RName* or *Resource*; we will qualify this with the terms "untimed" or "timed", where appropriate. We define

$$Resources \;\widehat{=}\; \mathbf{P}(Resource),$$

and write R for a typical element of *Resources*. We will need a couple of pieces of notation for dealing with resource sets: $\mathrm{Times}(R)$ returns the time domain of R, and $\mathrm{end}(R)$ returns the time of the last resource in R:

$$
\begin{array}{rcl}
\mathrm{Times}(R) & \widehat{=} & \{t \mid \exists\, r \bullet (t, r) \in R\}, \\
\mathrm{end}(R) & \widehat{=} & \max(\mathrm{Times}(R)).
\end{array}
$$

In order to define a scheduler for a program, we will insist that all constructs requiring a scheduling decision are labelled. In our current language, each primitive operation will require a particular set of timed resources (such as CPU), this set being calculable at compile time; we can therefore represent the resource requirements of such an agent by an element of type *Resources*.

We write *Lbl* for the set of labels, and write L for a typical label. We change the syntax of concrete TAM so that all constructs requiring a scheduling decision are labelled:

$$
\begin{array}{rl}
\mathcal{A} ::= & \mathsf{skip} \mid L :: \Delta t \mid L :: x := E \mid \\
& L :: (x, y) \leftarrow s \mid L :: E \rightarrow s \mid [t]\mathcal{A} \mid \mathcal{A} ; \mathcal{A} \mid \\
& \mathcal{A} \sqcap \mathcal{A}' \mid L :: \bigsqcup_{i \in I}{}_t\, g_i \Rightarrow \mathcal{A}_i \mid \mathcal{A} \mid \mathcal{A} \mid \mu_n\, \mathcal{A}\ .
\end{array}
$$

$$
\begin{aligned}
P_1 \;\widehat{=}\;& \mathsf{loop\ for}\ n\ \mathsf{period}\ p\ \bullet \\
& \quad SOV.v = open \wedge fuel.v = off \;\Rightarrow\; [t''_{del}]close \rightarrow SOV \\
& \quad \textstyle\bigsqcup_{t'_{stab}} \\
& \quad SOV.v = closed \wedge SAV.v = open \wedge fuel.v = on \;\Rightarrow\; [t''_{del}]open \rightarrow SOV, \\
P_2 \;\widehat{=}\;& \mathsf{loop\ for}\ n\ \mathsf{period}\ p\ \bullet \\
& \quad SAV.v = open \wedge (air.v = off \vee idling.v = true) \;\Rightarrow\; [t''_{del}]close \rightarrow SAV \\
& \quad \textstyle\bigsqcup_{t'_{stab}} \\
& \quad SAV.v = closed \wedge idling.v = false \wedge air.v = on \;\Rightarrow\; [t''_{del}]open \rightarrow SAV.
\end{aligned}
$$

Figure 4: The complete program.

We define a scheduler to be a function that takes a label and the time at which the label is reached in the program, and returns the set of (timed) resources on which the agent should be scheduled:

$$\phi \in Sch \,\widehat{=}\, Time \times Lbl \rightarrow Resources.$$

For example, consider the labelled assignment $L :: x := 3$. If this is encountered at time 7, with scheduler given by ϕ, then $\phi(7, L)$ gives the set of (timed) resources that will be used to execute the assignment. If the assignment takes two time units to complete then we might have that $\phi(7, L) = \{(9, r), (11, r)\}$, representing that the assignment will be scheduled on resource with resource name r at times 9 and 11: note that the times do not have to form a contiguous interval, so an operation can be interrupted.

We assume the existence of a function

$$\rho \;:\; Lbl \times RName \rightarrow \mathbf{Z}^+,$$

such that for each label L, $\rho(L, r)$ represents the amount of time it will take to execute the construct on resource r. We will ensure that

$$\phi(t, L) \subseteq Time \times \{r\} \;\Rightarrow\; \#\phi(t, L) = \rho(L, r),$$

i.e., if the construct is to be scheduled on resource name r, then ρ does indeed give the amount of time spent on the construct.

We will consider a *system* to be either a specification statement or a parallel composition of an agent with a scheduler:

$$
\begin{aligned}
\mathcal{S} \;&\in\; System, \\
\mathcal{S} \;&::=\; w : \Phi \;\big|\; \mathcal{A} \mid \phi.
\end{aligned}
$$

We will extend the TAM Logic Language in order to reason about resources. We will allow our predicates to include a distinguished free variable, R_{req}, representing the set of timed resources required by the agent. We will also add a boolean-valued variable fail which will be true precisely when an execution fails: this failure could be because (1) the agent tries to use a resource for two different purposes, simultaneously, or (2) a deadline is not met. This means that we are effectively assuming that all resources in $RName$ are always available; later we will extend the model to include consideration of the resources available to an agent, and to recognize that an agent will fail if it tries to use resources that are not available.

4.1 Semantic definitions

We will define a semantic function $\mathcal{R}$ giving the representation of an agent in this model:

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

Most of the semantic equations are simple extensions of the definitions in the previous section; in the following we concentrate our discussions upon resource and scheduling issues.

Our semantic definitions will ensure that all agents satisfy the following law, namely that resources are required only during the execution:

Law 4.1: $\mathcal{R}[\![\mathcal{A} \mid \phi]\!] \Rightarrow \mathrm{Times}(R_{req}) \subseteq [t_\alpha, t_\omega)$.

This means that most agents will terminate at time $t_\omega = \mathrm{end}(R_{req}) + 1$: the only exceptions will be those agents that deliberately idle after they finish using resources, for example in a construct such as $[t]\mathcal{A}$ where $\mathcal{A}$ terminates after fewer than t time units.

Our specification statement remains as before, except we insist that the agent does not fail:

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

The agent skip uses no resources:

$$
\begin{aligned}
\mathcal{R}[\![\mathsf{skip} \mid \phi]\!] \;\widehat{=}\;& t_\omega \geq t_\alpha \wedge stable(W, t_\alpha, t_\omega) \\
& \wedge R_{req} = \emptyset \wedge \neg\, \mathsf{fail}.
\end{aligned}
$$

The agent $L :: \Delta t$ will delay for t time units; if it is started at time t_α with scheduler ϕ then the delay is timed using the resources given by $R_{req} = \phi(t_\alpha, L)$.

$$\begin{aligned}\mathcal{R}[\![L :: \Delta t \mid \phi]\!] \ \widehat{=} \ & t_\omega = t_\alpha + t \wedge stable(W, t_\alpha, t_\omega) \\ & \wedge R_{req} = \phi(t_\alpha, L) \wedge \neg\, \mathsf{fail}.\end{aligned}$$

The assignment $L :: x := E$ is scheduled using the resources in $R_{req} = \phi(t_\alpha, L)$. The agent will terminate when it has used all the resources, i.e. at time $\mathrm{end}(R_{req}) + 1$.

$$\begin{aligned}&\mathcal{R}[\![L :: x := E \mid \phi]\!] \ \widehat{=} \\ &\quad stable(W \setminus x, t_\alpha, t_\omega) \wedge x@t_\omega = E@t_\alpha \\ &\quad \wedge R_{req} = \phi(t_\alpha, L) \wedge t_\omega = \mathrm{end}(R_{req}) + 1 \wedge \neg\, \mathsf{fail}.\end{aligned}$$

The definitions for shunt reads and shunt writes are similar:

$$\begin{aligned}&\mathcal{R}[\![L :: (x, y) \leftarrow s \mid \phi]\!] \ \widehat{=} \\ &\quad stable(W \setminus \{x, y\}, t_\alpha, t_\omega) \\ &\quad \wedge \exists\, t \in [t_\alpha, t_\omega) \bullet x@t_\omega = s@t.ts \wedge y@t = s@t.v \\ &\quad \wedge R_{req} = \phi(t_\alpha, L) \wedge t_\omega = \mathrm{end}(R_{req}) + 1 \wedge \neg\, \mathsf{fail}, \\ &\mathcal{R}[\![L :: E \rightarrow s \mid \phi]\!] \ \widehat{=} \\ &\quad stable(W \setminus s, t_\alpha, t_\omega) \\ &\quad \wedge \exists\, t \in (t_\alpha, t_\omega] \bullet stable(s, t_\alpha, t - 1) \\ &\qquad\qquad \wedge s@t = (t, E@t_\alpha) \\ &\qquad\qquad \wedge stable(s, t, t_\omega) \\ &\quad \wedge R_{req} = \phi(t_\alpha, L) \wedge t_\omega = \mathrm{end}(R_{req}) + 1 \wedge \neg\, \mathsf{fail}.\end{aligned}$$

The agent $[t]\mathcal{A}$ fails if $\mathcal{A}$ fails or if the deadline t is passed:

$$\begin{aligned}\mathcal{R}[\![[t]\mathcal{A} \mid \phi]\!] \ \widehat{=} \ & \exists\, t'_\omega : Time\, ; \mathsf{fail}' : Bool \bullet \\ & \quad \mathcal{R}[\![\mathcal{A} \mid \phi]\!][t'_\omega/t_\omega\, ,\ \mathsf{fail}'/\mathsf{fail}] \\ & \quad \wedge t_\omega = (t_\alpha + t) \sqcup t'_\omega \\ & \quad \wedge stable(W, t'_\omega, t_\omega) \\ & \quad \wedge (\mathsf{fail} \Leftrightarrow \mathsf{fail}' \vee t'_\omega > t_\alpha + t).\end{aligned}$$

$\mathcal{A}_1\ ; \mathcal{A}_2$ fails if either $\mathcal{A}_1$ or $\mathcal{A}_2$ fails; its resource requirements are the union of the resource requirements of the two components:

$$\begin{aligned}&\mathcal{R}[\![\mathcal{A}_1\ ; \mathcal{A}_2 \mid \phi]\!] \ \widehat{=} \\ &\quad \exists\, t \in [t_\alpha, t_\omega]\, ; \mathsf{fail}_1, \mathsf{fail}_2 : Bool; \\ &\qquad\qquad R_{req,1}, R_{req,2} : Resources \bullet \\ &\qquad \mathcal{R}[\![\mathcal{A}_1 \mid \phi]\!][t/t_\omega\, ,\ \mathsf{fail}_1/\mathsf{fail}\, ,\ R_{req,1}/R_{req}] \\ &\qquad \wedge \mathcal{R}[\![\mathcal{A}_2 \mid \phi]\!][t/t_\alpha\, ,\ \mathsf{fail}_2/\mathsf{fail}\, ,\ R_{req,2}/R_{req}] \\ &\qquad \wedge R_{req} = R_{req,1} \cup R_{req,2} \\ &\qquad \wedge (\mathsf{fail} \Leftrightarrow \mathsf{fail}_1 \vee \mathsf{fail}_2).\end{aligned}$$

As normal, nondeterministic choice is represented by disjunction:

$$\mathcal{R}[\![\mathcal{A} \sqcap \mathcal{A}' \mid \phi]\!] \ \widehat{=} \ \mathcal{R}[\![\mathcal{A} \mid \phi]\!] \vee \mathcal{R}[\![\mathcal{A}' \mid \phi]\!].$$

The agent $L :: \bigsqcup_{i \in I}{}_t\, g_i \Rightarrow \mathcal{A}_i$ evaluates the guards using the resources of $\phi(t_\alpha, L)$; if the evaluation does not meet the deadline t, then the agent fails.

$$\begin{aligned}&\mathcal{R}[\![L :: \textstyle\bigsqcup_{i \in I}{}_t\, g_i \Rightarrow \mathcal{A}_i \mid \phi]\!] \ \widehat{=} \\ &\quad \text{let} \quad J = \{i \in I \mid g_i@t_\alpha\} \quad \text{and} \quad R_g = \phi(t_\alpha, L) \\ &\qquad\quad \text{and} \quad t'_\alpha = (t_\alpha + t) \sqcup (\mathrm{end}(R_g) + 1) \quad \text{in} \\ &\quad \text{if} \quad J = \emptyset \\ &\quad \text{then} \quad t_\omega = t'_\alpha \wedge stable(W, t_\alpha, t_\omega) \\ &\qquad\quad \wedge R_{req} = R_g \wedge (\mathsf{fail} \Leftrightarrow t'_\alpha > t_\alpha + t) \\ &\quad \text{else} \quad stable(W, t_\alpha, t'_\alpha) \\ &\qquad\quad \wedge \exists\, j \in J\, ; \mathsf{fail}' : Bool\, ; R'_{req} : Resources \bullet \\ &\qquad\qquad \mathcal{R}[\![\mathcal{A}_j \mid \phi]\!][t'_\alpha/t_\alpha\, ,\ \mathsf{fail}'/\mathsf{fail}\, ,\ R'_{req}/R_{req}] \\ &\qquad\qquad \wedge R_{req} = R_g \cup R'_{req} \\ &\qquad\qquad \wedge (\mathsf{fail} \Leftrightarrow \mathsf{fail}' \vee t'_\alpha > t_\alpha + t).\end{aligned}$$

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.

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

As before, finite iteration is defined by

$$\begin{aligned}\mu_0\, \mathcal{A} \ &\widehat{=} \ \mathsf{skip}, \\ \mu_{n+1}\, \mathcal{A} \ &\widehat{=} \ \mathcal{A}\, ; \mu_n\, \mathcal{A}.\end{aligned}$$

We can define the obvious refinement relation between two systems:

Definition 4.2: $\mathcal{S} \sqsubseteq_{\mathcal{R}} \mathcal{S}'$ *iff* $\mathcal{R}[\![\mathcal{S}]\!] \Leftarrow \mathcal{R}[\![\mathcal{S}']\!]$.

System $\mathcal{S}$ is refined by system $\mathcal{S}'$ if the behaviour of $\mathcal{S}'$ is more fully defined than that of $\mathcal{S}$.

4.2 Limited resources

We can use this 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 set of timed resources available to the system. We then define a semantic function

$$\mathcal{L} \ : \ System \rightarrow TAMLL,$$

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:

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

In order to relate our scheduling models to our basic model, we need to define a relation between the two syntaxes: we write $\mathcal{A}$ *labels* $\mathcal{A}'$ if $\mathcal{A}$ can be obtained from $\mathcal{A}'$ by adding labels.

The following lemma relates the scheduling model to the functionality model. If we know how the scheduler behaves then we know more about the behaviour of the whole system; however, we must now consider the possibility of failure.

Lemma 4.3: For all agents $\mathcal{A}$ and $\mathcal{A}'$ and schedulers ϕ, if $\mathcal{A}$ *labels* $\mathcal{A}'$ then

$$\mathsf{fail} \vee \mathcal{F}[\![\mathcal{A}']\!] \;\Leftarrow\; \mathcal{L}[\![\mathcal{A} \mid \phi]\!].$$

This lemma can be proved by structural induction.

The following theorem will allow us to break down the development of a system into two stages, as in Figure 1: in the first stage we can use the standard TAM refinement calculus to derive an initial design; we can then use the scheduling-oriented model to derive a correct scheduler.

Theorem 4.4: Suppose we are given a set of available resources R_{av}. If the variables R_{av} and R_{req} do not appear free in the predicate Φ, and agent $\mathcal{A}'$ meets specification Φ, and agent $\mathcal{A}$ can be obtained from $\mathcal{A}'$ by adding labels, and the composition of $\mathcal{A}$ with the scheduler does not fail and meets specification Φ'; i.e.:

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

then the system meets the specification $\Phi \wedge \Phi'$:

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

Proof: We have:

$$\begin{array}{cl} & \mathcal{L}[\![\mathcal{A} \mid \phi]\!] \\ \Rightarrow & \langle\text{from the third assumption, and Lemma 4.3}\rangle \\ & \neg\,\mathsf{fail} \wedge \Phi' \wedge \mathcal{F}[\![\mathcal{A}']\!] \\ \Rightarrow & \langle\text{from the first assumption}\rangle \\ & \neg\,\mathsf{fail} \wedge \Phi' \wedge \mathit{stable}(W \setminus w, t_\alpha, t_\omega) \wedge \Phi \\ \equiv & \langle\text{definition}\rangle \\ & \mathcal{L}[\![w : \Phi \wedge \Phi']\!]. \end{array}$$

Hence $w : \Phi \wedge \Phi' \sqsubseteq_{\mathcal{L}} \mathcal{A} \mid \phi$. □

The following lemma will be very useful in proving the third of the above conditions (when $\Phi' \equiv \mathit{true}$):

Lemma 4.5: If the agents $\mathcal{A}_1$ and $\mathcal{A}_2$ with scheduler ϕ use sets of 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

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

4.3 Example

In this section we will design a scheduler for the aircraft engine controlling program we derived in Section 3.

We are going to implement our system on a single CPU r, which we assume is always available:

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

In order to obtain a scheduler for the program, we label the program constructs requiring scheduling decisions, namely the shunt writes and the evaluation of the alternation guards. We obtain the program $Q_1 \mid Q_2$, where Q_1 and Q_2 are defined in Figure 5.

Designing a scheduler ϕ then amounts to deciding when to perform each of the labelled constructs. From Theorem 4.4 it will be enough to ensure that

$$\mathcal{L}[\![Q_1 \mid Q_2 \mid \phi]\!] \;\Rightarrow\; \neg\,\mathsf{fail}.$$

We will assume that our CPU is fast enough to evaluate the guards of the alternations (labelled by L_1 and L_2) within 3 time units; and that it can perform the shunt writes (labelled by L_1', L_1'', L_2' and L_2'') within one time unit each; i.e.:

$$\rho(L_1, r) = \rho(L_2, r) = 3, \\ \rho(L_1', r) = \rho(L_1'', r) = \rho(L_2', r) = \rho(L_2'', r) = 1.$$

This means that at most 8 units of CPU-time will be used in each complete cycle, and so it seems sensible to try taking $p = 8$.

We will attempt to define a scheduler such that during each 8 time unit cycle, the guards are evaluated during the first 6 time units, and the shunt writes are performed during the final 2 time units. Thus we will take the following values for the time parameters:

$$p = 8, \qquad t'_{stab} = 6, \qquad t''_{del} = 2.$$

More definitely, our scheduler ϕ will ensure that Q_1's guards are evaluated during the first three time units,

$$Q_1 \mathrel{\widehat{=}} \textsf{loop for } n \textsf{ period } p \bullet$$
$$L_1 :: \begin{pmatrix} SOV.v = open \wedge fuel.v = off \;\Rightarrow\; [t''_{del}]\; L'_1 :: close \rightarrow SOV \\ \bigsqcup_{t'_{stab}} \\ SOV.v = closed \wedge SAV.v = open \wedge fuel.v = on \;\Rightarrow\; [t''_{del}]\; L''_1 :: open \rightarrow SOV \end{pmatrix},$$
$$Q_2 \mathrel{\widehat{=}} \textsf{loop for } n \textsf{ period } p \bullet$$
$$L_2 :: \begin{pmatrix} SAV.v = open \wedge (air.v = off \vee idling.v = true) \;\Rightarrow\; [t''_{del}]\; L'_2 :: close \rightarrow SAV \\ \bigsqcup_{t'_{stab}} \\ SAV.v = closed \wedge idling.v = false \wedge air.v = on \;\Rightarrow\; [t''_{del}]\; L''_2 :: open \rightarrow SAV \end{pmatrix}.$$

Figure 5: The labelled program.

then Q_2's guards are evaluated during the next three time units, then Q_1 performs its shunt write, and finally Q_2 performs its shunt write. From Lemma 4.5 it will be enough to ensure:

$$\begin{aligned} &\mathcal{R}[\![Q_1 \mid \phi]\!] \Rightarrow \\ &\quad R_{req} \subseteq \{(t,r) \mid (t - t_\alpha) \bmod 8 \in \{0,1,2,6\}\} \\ &\quad \wedge \neg\, \mathsf{fail}, \\ &\mathcal{R}[\![Q_2 \mid \phi]\!] \Rightarrow \\ &\quad R_{req} \subseteq \{(t,r) \mid (t - t_\alpha) \bmod 8 \in \{3,4,5,7\}\} \\ &\quad \wedge \neg\, \mathsf{fail}. \end{aligned}$$

If we can achieve this then the system will not fail, because the resource requirements will be disjoint.

It is obvious that the mth iteration starts at time $t_\alpha + 8m$, and so the labels L_1 and L_2 are encountered at times $t_\alpha + 8m$ (for $m \in [0,n)$), and labels L'_1, L''_1, L'_2 and L''_2 are encountered at times $t_\alpha + 8m + 6$ (for $m \in [0,n)$). Thus we will schedule Q_1's alternation immediately and delay Q_2's for 3 time units:

$$\begin{aligned} \phi(t, L_1) &= \{(t,r), (t+1,r), (t+2,r)\}, \\ \phi(t, L_2) &= \{(t+3,r), (t+4,r), (t+5,r)\}. \end{aligned}$$

And we will schedule Q_1's shunt writes immediately, and delay Q_2's for one time unit:

$$\begin{aligned} \phi(t, L'_1) = \{(t,r)\}, &\quad \phi(t, L''_1) = \{(t,r)\}, \\ \phi(t, L'_2) = \{(t+1,r)\}, &\quad \phi(t, L''_2) = \{(t+1,r)\}. \end{aligned}$$

Then it is easily seen that the components meet the above requirements.

Finally, we can calculate values for the time parameters:

$$t_{stab} = 13, \qquad t_{del} = 15.$$

In fact, our system will also meet the specification for any larger values of these time parameters.

5 A model with dynamic scheduling

In this section we extend our model so as to be able to reason about dynamic scheduling, where scheduling may depend upon how different parts of the program have progressed so far. The idea is that we will allow the scheduling decisions to be a function of the labels encountered so far. We therefore need to keep track of the times at which labels were encountered, and so extend our predicate language to include a free variable χ, representing the labels or checkpoints encountered:

$$\chi \in CheckPts \mathrel{\widehat{=}} \mathbf{P}(Time \times Lbl).$$

We define a dynamic scheduler to be a function from the check points so far encountered to a fixed scheduler:

$$\psi \in DSch \mathrel{\widehat{=}} CheckPts \rightarrow Sch.$$

For example, suppose one parallel component contains a critical section between labels L_1 and L_2, such that no other agents should be scheduled while this component is in the critical section. Then we can define an appropriate dynamic scheduler. For example, if the agent labelled by L should, subject to the above condition, be scheduled as soon as possible on the resource with resource name r, for one time unit, then

$$\psi(\chi)(t_\alpha, L) = \begin{cases} \{(t_\alpha, r)\}, & \text{if } \#\{t \mid (t, L_1) \in \chi \wedge t \leq t_\alpha\} \\ & \qquad = \#\{t \mid (t, L_2) \in \chi \wedge t \leq t_\alpha\} \\ \{(t'_\alpha, r)\}, & \text{otherwise} \\ & \text{where } t'_\alpha = \min\{t > t_\alpha \mid (t, L_2) \in \chi\}. \end{cases}$$

If the label is encountered at t_α, then if the critical section is left as often as it is entered before t_α, then the agent should be scheduled immediately; otherwise it should be scheduled as soon as the critical section is left.

We are going to consider only *non-clairvoyant* schedulers:

Definition 5.1: We say that a scheduler ψ is *non-clairvoyant* iff, whenever two sets of checkpoints χ_1 and χ_2 agree up until time t, i.e.,

$$\{(t', L') \in \chi_1 \mid t' \leq t\} = \{(t', L') \in \chi_2 \mid t' \leq t\},$$

scheduling decisions made before time t should be the same whether they are based on χ_1 or χ_2: when $\psi(\chi_1)$ and $\psi(\chi_2)$ are applied to label L at time $t' \leq t$, the two sets of resources returned should agree up until time t:

$$\begin{array}{l} \forall L \in Lbl \bullet \forall t' \leq t \bullet \\ \quad \{(t'', r) \in \psi(\chi_1)(t', L) \mid t'' \leq t\} \\ \quad\quad = \{(t'', r) \in \psi(\chi_2)(t', L) \mid t'' \leq t\}. \end{array}$$

The behaviour of an agent will depend upon the check points encountered elsewhere in the system, so we define a semantic function:

$$\mathcal{D} \ : System \rightarrow CheckPts \rightarrow TAMLL,$$

such that $\mathcal{D}[\![\mathcal{A} \mid \psi]\!]\chi'$ describes the behaviour of the agent $\mathcal{A}$, with dynamic scheduler ψ, when checkpoints χ' are encountered elsewhere in the system. In most cases, the agent will be scheduled according to the fixed scheduler $\psi(\chi')$.

Initially we consider a model where there are unlimited resources available, analogous to the first model of Section 4; we can then extend this to consider limited resources.

We will take care to ensure that our definitions are not circular. Scheduling decisions taken in one part of the system at time t depend upon the labels encountered elsewhere in the system up to time t (because of Definition 5.1). The labels encountered elsewhere in the system up to time t depend upon only the scheduling decisions taken *strictly before* t. Hence the scheduling decisions taken in one part of the system at time t depend upon only the scheduling decisions taken elsewhere strictly before t, and so there is no circularity.

Our specification statement takes the normal form:

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

The semantic equations for the labelled atomic agents must be adapted so that the scheduling depends upon the labels encountered elsewhere. The definition must also be extended to record the checkpoints encountered here. For example:

$$\begin{array}{rl} \mathcal{D}[\![L :: x := E \mid \psi]\!]\chi' \ \widehat{=} & \mathcal{R}[\![L :: x := E \mid \psi(\chi')]\!] \\ & \wedge\ \chi = \{(t_\alpha, L)\}. \end{array}$$

The agent is scheduler according to the fixed scheduler $\psi(\chi')$; the label L is encountered at time t_α.

The semantic definition for $\mathcal{A}_1 \ ; \mathcal{A}_2$ is similarly adapted:

$$\begin{array}{l} \mathcal{D}[\![\mathcal{A}_1 \ ; \mathcal{A}_2 \mid \psi]\!]\chi' \ \widehat{=} \\ \quad \exists\, t \in [t_\alpha, t_\omega] \ ; \mathsf{fail}_1, \mathsf{fail}_2 : Bool; \\ \quad\quad R_{req,1}, R_{req,2} : Resources \ ; \chi_1, \chi_2 : CheckPts \ \bullet \\ \quad\quad\quad \mathcal{D}[\![\mathcal{A}_1 \mid \psi]\!](\chi' \cup \chi_2) \begin{bmatrix} t/t_\omega \ , \ \mathsf{fail}_1/\mathsf{fail} \ , \\ R_{req,1}/R_{req} \ , \ \chi_1/\chi \end{bmatrix} \\ \quad\quad\quad \wedge\ \mathcal{D}[\![\mathcal{A}_2 \mid \psi]\!](\chi' \cup \chi_1) \begin{bmatrix} t/t_\alpha \ , \ \mathsf{fail}_2/\mathsf{fail} \ , \\ R_{req,2}/R_{req} \ , \ \chi_2/\chi \end{bmatrix} \\ \quad\quad\quad \wedge\ R_{req} = R_{req,1} \cup R_{req,2} \wedge (\mathsf{fail} \Leftrightarrow \mathsf{fail}_1 \vee \mathsf{fail}_2) \\ \quad\quad\quad \wedge\ \chi = \chi_1 \cup \chi_2. \end{array}$$

The checkpoints encountered outside of $\mathcal{A}_2$ include the checkpoints χ_1 of $\mathcal{A}_1$, and so our definition allows scheduling decisions inside $\mathcal{A}_2$ to depend on these. Similarly, the checkpoints encountered outside $\mathcal{A}_1$ include the checkpoints χ_2 of $\mathcal{A}_2$; however, because we are dealing with non-clairvoyant schedulers, these make no difference to scheduling decisions inside $\mathcal{A}_1$, and so we could equally have written $\mathcal{D}[\![\mathcal{A}_1 \mid \psi]\!]\chi'$. The checkpoints in $\mathcal{A}_1 ; \mathcal{A}_2$ are simply the union of the checkpoints in $\mathcal{A}_1$ and $\mathcal{A}_2$

The definition of parallel composition is a similar adaptation of the definition in the model for fixed scheduling:

$$\begin{array}{l} \mathcal{D}[\![(\mathcal{A}_1 \mid \mathcal{A}_2) \mid \psi]\!]\chi' \ \widehat{=} \\ \quad \exists\, t_1, t_2 : Time \ ; \mathsf{fail}_1, \mathsf{fail}_2 : Bool; \\ \quad\quad R_{req,1}, R_{req,2} : Resources \ ; \chi_1, \chi_2 : CheckPts \ \bullet \\ \quad\quad\quad \mathcal{D}[\![\mathcal{A}_1 \mid \psi]\!](\chi' \cup \chi_2) \begin{bmatrix} t_1/t_\omega \ , \ \mathsf{fail}_1/\mathsf{fail} \ , \\ R_{req,1}/R_{req} \ , \ \chi_1/\chi \end{bmatrix} \\ \quad\quad\quad \wedge\ \mathcal{D}[\![\mathcal{A}_2 \mid \psi]\!](\chi' \cup \chi_1) \begin{bmatrix} t_2/t_\omega \ , \ \mathsf{fail}_2/\mathsf{fail} \ , \\ R_{req,2}/R_{req} \ , \ \chi_2/\chi \end{bmatrix} \\ \quad\quad\quad \wedge\ stable(W_1, t_1, t_\omega) \wedge stable(W_2, t_2, t_\omega) \\ \quad\quad\quad \wedge\ t_\omega = t_1 \sqcup t_2 \wedge R_{req} = R_{req,1} \cup R_{req,2} \\ \quad\quad\quad \wedge\ \chi = \chi_1 \cup \chi_2 \\ \quad\quad\quad \wedge\ (\mathsf{fail} \Leftrightarrow \mathsf{fail}_1 \vee \mathsf{fail}_2 \vee R_{req,1} \cap R_{req,2} \neq \emptyset). \end{array}$$

As in Section 4, we can use this model to define another semantic model that models the fact that there may be limited resources available. We define a semantic function

$$\mathcal{E} \ : \ System \rightarrow TAMLL,$$

by

$$\begin{array}{rl} \mathcal{E}[\![\mathcal{S}]\!] \ \widehat{=} & \exists\, \mathsf{fail}' \bullet \mathcal{D}[\![\mathcal{S}]\!]\emptyset[\mathsf{fail}'/\mathsf{fail}] \\ & \wedge\ (\mathsf{fail} \Leftrightarrow \mathsf{fail}' \vee R_{req} \not\subseteq R_{av}). \end{array}$$

Here we are considering a system at the top level, and so we assume that there are no checkpoints encountered elsewhere in the system. Refinement in this model is defined in the obvious way:

$$\mathcal{S} \sqsubseteq_\mathcal{E} \mathcal{S}' \ \ iff \ \ \mathcal{E}[\![\mathcal{S}]\!] \Leftarrow \mathcal{E}[\![\mathcal{S}']\!].$$

The following lemma and theorem are analogous to Lemma 4.3 and Theorem 4.4.

Lemma 5.2: For all agents $\mathcal{A}$ and $\mathcal{A}'$ and schedulers ψ, if $\mathcal{A}$ *labels* $\mathcal{A}'$ then:

$$\mathsf{fail} \lor \mathcal{F}[\![\mathcal{A}']\!] \;\Leftarrow\; \mathcal{E}[\![\mathcal{A} \mid \psi]\!].$$

Theorem 5.3: Suppose we are given a set of available resources R_{av}. If the variables R_{av} and R_{req} do not appear free in the predicate Φ, and agent $\mathcal{A}'$ meets specification Φ, and agent $\mathcal{A}$ can be obtained from $\mathcal{A}'$ by adding labels, and the composition of $\mathcal{A}$ with the scheduler does not fail and meets specification Φ'; i.e.:

$$\begin{array}{rcl} w : \Phi & \sqsubseteq_{\mathcal{F}} & \mathcal{A}', \\ \mathcal{A} & \mathit{labels} & \mathcal{A}', \\ \mathcal{E}[\![\mathcal{A} \mid \psi]\!] & \Rightarrow & \neg\, \mathsf{fail} \land \Phi', \end{array}$$

then the system meets the specification $\Phi \land \Phi'$:

$$w : \Phi \land \Phi' \;\sqsubseteq_{\mathcal{E}}\; \mathcal{A} \mid \psi.$$

This theorem allows us to consider the questions of functionality and (dynamic) scheduling separately.

6 Conclusions

In this paper we have developed a hierarchy of models for reasoning about allocation of resources and scheduling in real-time distributed systems. We have represented agents by predicates describing their possible behaviours. In our first model the predicates talked only about the agents' functionality. In the second model we extended this to talk about resource allocation and fixed (i.e. compile time) scheduling. We represented a scheduler as a function that, given a label and the time at which the label is encountered, returns the set of timed resources upon which the agent should be scheduled: the semantic model tells us whether this scheduler is correct in the sense that no two agents are scheduled on the same resource simultaneously, and that all deadlines are met. Finally, we extended the model to deal with dynamic scheduling, where scheduling decisions are allowed to depend upon progress made elsewhere in the system.

The work in this paper so far has presented a fairly basic model for scheduling. In this final section we briefly sketch ways in which the scheduling model could be extended to cover more phenomena.

6.1 Variable duration constructs

The language we have considered so far does not include constructs for introducing new variables or shunts. We may add these to the language:

$$\mathcal{A} \;::=\; \ldots \;\mid\; (x : T)\mathcal{A} \;\mid\; (s : T)\mathcal{A}.$$

$(x : T)\mathcal{A}$ declares x to be a new local variable of type T within $\mathcal{A}$; similarly, $(s : T)\mathcal{A}$ declares s to be a new shunt holding a $\mathit{Time} \times T$ pair.

It is easy to give definitions for these in the functionality model. For example:

$$\begin{array}{l} \mathcal{F}[\![(x : T)\mathcal{A}]\!] \mathrel{\widehat{=}} \\ \qquad \exists\, v : \mathit{Time} \twoheadrightarrow T \bullet \mathcal{F}[\![\mathcal{A}]\!][v(t)/x@t \mid t \in [t_\alpha, t_\omega]]. \end{array}$$

The value of variable x at time t is given by $v(t)$; the values are hidden using an existential quantification.

Adding these constructs to the scheduling model is slightly harder, because it is not normally feasible to calculate at compile time for how long the variable is going to be needed. Our scheduling model therefore needs to be able to handle two sorts of constructs: (1) *fixed duration constructs*, such as assignments, shunt reads, shunt writes, and delays, which we can handle as in Section 4; and (2) *variable duration constructs*, such as local variables, which we will handle by defining a function $\alpha : \mathit{Time} \twoheadrightarrow \mathit{RName}$ such that $\alpha(t)$ gives the name of the (untimed) resource on which the variable should be held at time t; note that this approach allows the variable to be held in different locations at different times, for example if some memory is swapped into secondary store.

We will label the two different sorts of constructs with labels from two disjoint sets. We write $\mathit{Lbl}_{\mathit{fixed}}$ for the set of labels for fixed duration constructs and $\mathit{Lbl}_{\mathit{variable}}$ for the set of labels for variable duration constructs.

We define a scheduler to be a function that takes a label and the time at which the label is reached in the program: for labels corresponding to fixed duration constructs, the scheduler returns the set of resources on which the agent should be scheduled, as in Section 4; for labels corresponding to variable duration constructs, the scheduler returns a function α from time to resource names, such that $\alpha(t)$ gives the resource that should be used at time t:

$$\begin{array}{rl} \phi \in \mathit{Sch} \mathrel{\widehat{=}} & (\mathit{Time} \times \mathit{Lbl}_{\mathit{fixed}} \to \mathit{Resources}) \\ & \cup (\mathit{Time} \times \mathit{Lbl}_{\mathit{variable}} \to \mathit{Time} \twoheadrightarrow \mathit{RName}). \end{array}$$

If we want to distinguish between processing and memory resources, then we can partition RName into two sets, RProc and RMem, and define schedulers appropriately:

$$\begin{array}{rl} \mathit{Sch} \mathrel{\widehat{=}} & (\mathit{Time} \times \mathit{Lbl}_{\mathit{fixed}} \to \mathbf{P}(\mathit{Time} \times \mathit{RProc})) \\ & \cup (\mathit{Time} \times \mathit{Lbl}_{\mathit{variable}} \to \mathit{Time} \twoheadrightarrow \mathit{RMem}). \end{array}$$

We can then give semantics to the local variable construct as follows:

$$
\begin{array}{l}
\mathcal{R}[\![L :: (x : T)\mathcal{A} \mid \phi]\!] \mathrel{\widehat{=}} \\
\quad \exists\, v : \mathit{Time} \nrightarrow T \,;\, \mathsf{fail}' : \mathit{Bool} \,;\, R'_{req} : \mathit{Resources} \bullet \\
\qquad \text{let} \quad R_x = \{(t, \phi(t_\alpha, L)(t)) \mid t \in [t_\alpha, t_\omega)\} \quad \text{in} \\
\qquad \mathcal{R}[\![\mathcal{A} \mid \phi]\!][v(t)/x@t \mid t \in [t_\alpha, t_\omega]] \\
\qquad\qquad [\mathsf{fail}'/\mathsf{fail}, R'_{req}/R_{req}] \\
\qquad \wedge\, R_{req} = R'_{req} \cup R_x \\
\qquad \wedge\, (\mathsf{fail} \Leftrightarrow \mathsf{fail}' \vee R_x \cap R'_{req} \neq \emptyset).
\end{array}
$$

The variable is scheduled on resource $\phi(t_\alpha, L)(t)$ at time t, for each $t \in [t_\alpha, t_\omega)$.

6.2 Generalizing dynamic schedulers

We have allowed our dynamic schedulers to base their scheduling decisions upon the checkpoints encountered in the program; we could also allow scheduling decisions to depend upon the state of variables or shunts throughout the execution. For example, if we decided that we wanted our scheduling decisions to depend upon the successive values taken by the shunt $s : T$, then we could define a dynamic scheduler to take two arguments, a set of checkpoints and the values held by the shunt, and return a fixed scheduler:

$$\psi \in \mathit{DSch} \mathrel{\widehat{=}} \mathit{CheckPts} \times (\mathit{Time} \nrightarrow \mathit{Time} \times T) \rightarrow \mathit{Sch}.$$

(As before, we consider only non-clairvoyant schedulers.) For example, the semantic equation for assignment becomes

$$
\begin{array}{rcl}
\mathcal{D}[\![L :: x := E \mid \psi]\!]\chi' & \mathrel{\widehat{=}} & \mathcal{R}[\![L :: x := E \mid \psi(\chi', s)]\!] \\
& & \wedge\, \chi = \{(t_\alpha, L)\}.
\end{array}
$$

Implementing this would require that the successive values held in s be remembered throughout the rest of the execution.

Many schedulers use the concept of priorities. It is possible to code up priorities using the dynamic scheduling model, as shown in the example at the start of Section 5. Alternatively, it may be possible to extend the model to reason about priorities directly.

6.3 Soft deadlines

The deadlines in this paper have all been *hard*, in the sense that failure to meet a deadline is considered catastrophic. Alternatively, we could consider *soft* deadlines, where some penalties are imposed for failure to meet a deadline; the model should tell us how great the penalty would be for a given scheduler. We could do this by allowing our predicates to include a variable, $\mathsf{penalty}$, representing the value of the penalty; for example, if we wanted our penalty to be the sum of the amounts by which deadlines were missed, the semantic equation for the deadline operator would be:

$$
\begin{array}{l}
\mathcal{R}[\![[t]\mathcal{A} \mid \phi]\!] \mathrel{\widehat{=}} \\
\quad \exists\, t'_\omega : \mathit{Time} \,;\, \mathsf{penalty}' : \mathbf{N} \bullet \\
\qquad \mathcal{R}[\![\mathcal{A} \mid \phi]\!][t'_\omega/t_\omega \,,\, \mathsf{penalty}'/\mathsf{penalty}] \\
\qquad \wedge\, t_\omega = (t_\alpha + t) \sqcup t'_\omega \wedge \mathit{stable}(W, t'_\omega, t_\omega) \\
\qquad \wedge\, \mathsf{penalty} = \mathsf{penalty}' + (t_\omega - t_\alpha - t \sqcup 0).
\end{array}
$$

The term $t_\omega - t_\alpha - t \sqcup 0$ represents the amount by which this deadline is missed.

6.4 Using the model

In this paper we tackled our example by firstly using the functionality model to derive a program, and then using the scheduling-oriented model to develop a scheduler. In [Low94] we considered the same problem, but also used the scheduling-oriented model to develop a program and scheduler simultaneously. The second solution was better, in the sense that the resulting system met tighter deadlines, but the calculations to produce it were harder. We were able to consider the two aspects together only because we could see where the derivation was going, which enabled us to make scheduling decisions early in the derivation. We therefore suggest that the development process should always be split into two phases unless the deadlines are particularly tight.

Another case study appears in [LZ95]. This paper develops useful specification techniques and proof rules for calculating correct programs and schedulers.

Work is currently being carried out to develop a tool for analyzing TAM agents; it is hoped that this will eventually include consideration of resource issues, so as to make reasoning about scheduling easier.

Acknowledgements

I would like to thank He Jifeng, Hussein Zedan, Tony Hoare, and the anonymous referees for their useful comments.

References

[CH88] R. Cleaveland and M. Hennessy. Priorities in process algebras. In *Proc. 3rd Symposium on Logic in Computer Science, Edinburgh*, 1988.

[DRRS95] J. Davies, G. M. Reed, A. W. Roscoe, and S. A. Schneider. *Real Time CSP*. Prentice Hall, 1995. Forthcoming.

[DS95] Jim Davies and Steve Schneider. A brief history of timed CSP. *Theoretical Computer Science*, 138, 1995.

[GL89] R. Gerber and I. Lee. Communicating shared resources: A model for distributed real-time systems. In *IEEE Real-Time Systems Symposium*, 1989.

[GL90] R. Gerber and I. Lee. A calculus for communicating shared resources. In *Concur '90, LNCS 458*. Springer Verlag, 1990.

[GL92] R. Gerber and I. Lee. Specification and analysis of resource-bound real-time systems. In *Real-Time: Theory in Practice, LNCS 600*, pages 371–396. Springer Verlag, 1992.

[HR90] M. Hennessy and T. Regan. A temporal process algebra. Technical Report 2-90, University of Sussex, 1990.

[Jac90] David Jackson. The specification of aircraft engine software using Timed CSP. Master's thesis, Oxford University, 1990.

[Low94] Gavin Lowe. Formal development of aircraft control software: A case study in the specification, design and scheduling of a real-time system. Technical Report PRG-TR-15-94, Oxford University Computing Laboratory, 1994.

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

[Mil89] Robin Milner. *Communication and Concurrency*. Prentice Hall International, 1989.

[MT90] F. Moller and C. Tofts. A temporal calculus of communicating systems. In *LNCS 458, Concur '90*. Springer Verlag, 1990.

[RR86] G. M. Reed and A. W. Roscoe. A timed model for CSP. In *Theoretical Computer Science 58*, 1986.

[SZ92] D. Scholefield and H. Zedan. The refinement of real-time systems. In *Fourth European Workshop on Real-Time Systems*. Athens, Greece, IEEE Computer Society Press, June 1992.

[SZ93] D. Scholefield and H. Zedan. A proof system for real-time programs. Technical Report YCS 210, University of York, 1993.

[SZJ94] D. Scholefield, H. Zedan, and He Jifeng. A specification oriented semantics for the refinement of real-time system. *Theoretical Computer Science*, 131(1), 1994.

[Yi90] Wang Yi. Real-time behaviour of asynchronous agents. In *Proceedings of Concur '90, LNCS 458*. Springer Verlag, 1990.

A Refinement rules

In this section we give the refinement rules that we used in this paper.

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

Rule 2 (Parallel composition): If $w \cap w' = \emptyset$, and t_ω does not appear free in Φ or Φ', then

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

Rule 3 (Periodic loop): For $p \in \mathbf{Z}^+$ (the period of the loop):

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

Rule 4 (Alternation): If

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

then

$$w : \Phi \ \sqsubseteq \ \bigsqcup_{i \in I}{}_t \Big(g_i \Rightarrow w : \Phi_i\Big).$$

Rule 5 (Deadline):

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

Rule 6 (Shunt write): If E is an expression, and $s \in w$, then

$$w : \mathsf{write}(s, E @ t_\alpha, t_\alpha, t_\omega) \ \sqsubseteq \ E \rightarrow s.$$