Programming Research Group

FORMAL DEVELOPMENT OF AIRCRAFT CONTROL SOFTWARE: A CASE STUDY IN THE SPECIFICATION, DESIGN AND SCHEDULING OF A REAL-TIME SYSTEM

Gavin Lowe

PRG-TR-15-94

Oxford University Computing Laboratory
Wolfson Building, Parks Road, Oxford OX1 3QD

Formal Development of Aircraft Control Software: A Case Study in the Specification, Design and Scheduling of a Real-Time System[1]

Gavin Lowe

October 1994

Abstract

In this paper we consider a simple example of embedded software taken from the aircraft industry [Rol88]. We use the formal models for scheduling analysis developed in [Low94] to specify, design and schedule a program that controls part of the start-up of an aircraft engine.

1 Introduction

Many formalisms exist for specifying, designing and verifying real-time systems, for example, the Temporal Agent Model (TAM) [SZ92, SZJ94], Timed CSP [DS94, DRRS94] and Timed CCS [HR90, MT90, Yi90]. However, these formalisms have not dealt with the important problems of resource allocation and scheduling. The formalisms have adopted the *unbounded parallelism hypothesis*, i.e. that sufficient resources are always available. However, in real systems this is rarely true.

In [Low94], I developed an extended model of TAM which allows one to reason about resources and scheduling. In this paper we apply this model to a medium-sized case study, in order to assess how easy the model is to use and to investigate whether any changes to the model are desirable. We apply the model to a problem concerning part of the start-up routine for an aircraft engine [Rol88]; this problem was previously studied in [Jac90] as a case study in Timed CSP.

In the next section we give an overview of the Temporal Agent Model and the scheduling model from [Low94]. In section 3 we give an informal description of the problem, and produce a formal specification. We develop a solution in

[1]Further copies of this Technical Report may be obtained from the Librarian (Oxford University Computing Laboratory, Programming Research Group, Wolfson Building, Parks Road, Oxford OX1 3QD, England, telephone: +44-1865-273837, e-mail: `library@comlab.ox.ac.uk`) or by anonymous ftp from `ftp.comlab.ox.ac.uk`, filename `pub/Documents/techpapers/Gavin.Lowe/aircraft.ps.gz`.

section 4; the development process makes use of a theorem proved in [Low94] which allows us to split the development process into two stages: in the first stage we use the standard TAM model to develop an initial design without a scheduler; in the second stage we design and verify a scheduler. In section 5 we adopt a different approach: by considering the program design and scheduler together we derive a system that meets a stronger specification with tighter deadlines; the cost of the better solution is that the derivation is harder. In section 6 we sum up and review the lessons we have learnt from this case study.

2 The Temporal Agent Model

The Temporal Agent Model (TAM) is a wide spectrum language for modelling real-time, distributed computer systems. The TAM language includes most of the standard features of a real-time programming language, such as parallel composition, deadlines and communication. The language also includes a specification statement; a number of algebraic laws allow such a specification to be refined into concrete TAM via provably correct transformations.

A TAM program is a collection of parallel agents, each with their own set of variables. Agents communicate via a form of asynchronous channel called a *shunt*; a shunt can be thought of as a location in shared memory, holding a pair: a value, and a time-stamp giving the last time at which the shunt was updated. Each shunt is owned by a single agent that can write values to it; all agents can then read this value.

In this paper we will consider the subset of the TAM language defined by the BNF expression:

$$\begin{array}{c} \mathcal{A} \ ::= \ w : \Phi \ \mid \ \mathsf{skip} \ \mid \ \Delta t \ \mid \ x := E \ \mid \ E \rightarrow s \ \mid \ (x, y) \leftarrow s \ \mid \\ [t]\mathcal{A} \ \mid \ \mathcal{A} \, ; \mathcal{A} \ \mid \ \bigsqcup_{t \atop i \in I} g_i \Rightarrow \mathcal{A}_i \ \mid \ \mathcal{A} \mid A \ \mid \ \mu \, X_n \bullet \mathcal{A} \ , \end{array}$$

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

$w : \Phi$ is a specification statement: it specifies that only the variables in the frame w may be changed by the program, and that all executions must satisfy the predicate Φ. In [SZ92, SZJ94], refinement rules are given that allow a specification in this form to be refined into concrete TAM. Some of these rules are reproduced in appendix A.

The agent skip terminates immediately, whereas Δt terminates after t seconds. The agent $x := E$ evaluates E, storing the result in variable x. $E \rightarrow s$ evaluates E, and writes the result to shunt s, time-stamping it with the time of the write. The agent $(x, y) \leftarrow s$ reads shunt s, storing the time-stamp of the last update in x, and the value in y.

Deadlines are specified by the construct $[t]\mathcal{A}$, which executes $\mathcal{A}$ with the proviso that the execution must last t seconds. If $\mathcal{A}$ terminates before t seconds then the agent idles to fill the interval[2]; if the execution fails to meet the deadline then we consider the agent to have failed.

$\mathcal{A}$; $\mathcal{A}$ represents a sequential composition, with the second agent being started at the termination time of the first. Alternation is modelled by the agent $\bigsqcup_{t\,i \in I} g_i \Rightarrow \mathcal{A}_i$: each of the guards g_i is evaluated and if some are true, a corresponding $\mathcal{A}_i$ is executed; if all the g_i evaluate to false then the agent simply terminates; the guards must be evaluated within t seconds and it is assumed that the run time environment will idle to fill this period.

The agent $\mathcal{A} \mid \mathcal{A}'$ executes $\mathcal{A}$ and $\mathcal{A}'$ in parallel, terminating when both agents have terminated. The agent $\mu\, X_n \bullet \mathcal{A}$ represents a finite iteration, executing $\mathcal{A}$ n times.[3]

It will be convenient to introduce the derived operator $Period(p, n, \mathcal{A})$ which executes $\mathcal{A}$ n times with a period of p:

$$Period(p, n, \mathcal{A}) \mathrel{\widehat{=}} \mu\, X_n \bullet [p]\mathcal{A}.$$

In [SZJ94] a semantics was given for this language, representing an agent 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. Φ may also include terms of the form $s@t$ to refer to the contents of shunt s at time t, and similarly $x@t$ to refer to the value held in variable x. We write $s.ts@t$ for the time-stamp of s at t, and $s.v@t$ for the value. We will also write $E@t$ to represent the value of expression E where all the variables are taken at time t.

We can define a semantic function

$$\mathcal{P} : TAM \rightarrow TAMLL,$$

such that $\mathcal{P}[\![\mathcal{A}]\!]$ gives the predicate describing the functionality of agent $\mathcal{A}$. For example,

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

Here we take W to be the alphabet of the agent. The clause $t_\omega > t_\alpha$ specifies that the assignment takes some positive amount of time. The clause $stable(W \setminus x, t_\alpha, t_\omega)$ specifies that all variables other than x must remain unchanged during the execution of this assignment. The clause $x@t_\omega = E@t_\alpha$ specifies that the final value of x must equal the initial value of the expression E.

Refinement between agents can be defined by

$$\mathcal{A} \sqsubseteq \mathcal{A}' \;\; \textit{iff} \;\; \mathcal{P}[\![\mathcal{A}]\!] \Leftarrow \mathcal{P}[\![\mathcal{A}']\!].$$

$\mathcal{A}$ is refined by $\mathcal{A}'$ if $\mathcal{A}'$ is more fully defined than $\mathcal{A}$. The semantic definitions can be used to define a number of refinement rules; some of these are given in appendix A.

[2]This is a change from the earlier paper, where $[t]\mathcal{A}$ terminated as soon as $\mathcal{A}$ terminated, even if this was before t seconds had elapsed; our experience of doing this case study suggests that the construct given here is more useful.

[3]This is another slight change from the previous paper, where we allowed the agent $\mathcal{A}$ to contain the variable X, representing a recursive call.

Scheduling oriented models 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):

$$Resource \mathrel{\widehat{=}} Time \times RName.$$

We altered the syntax so that all constructs requiring a scheduling decision are labelled. We then defined a scheduler to be a function that takes a label and the time at which the program reaches the label, and returns the set of resources upon which the corresponding agent should be scheduled:

$$Sch \mathrel{\widehat{=}} Time \times Lbl \to \mathbb{P}(Resource).$$

For example, if the labelled assignment $L :: x := 3$ is encountered at time 7, with scheduler given by ϕ, and $\phi(7, L) = \{(9, r), (11, r)\}$, 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 restrict our syntax to consider only parallel compositions of sequential agents; writing $\mathcal{A}$ for a sequential agent, $\mathcal{S}$ for a system:

$$\begin{aligned}
\mathcal{A} \ ::= \ & w : \Phi \mid \mathsf{skip} \mid \Delta t \mid L :: x := E \mid L :: E \to s \mid L :: (x, y) \leftarrow s \mid \\
& [t]\mathcal{A} \mid \mathcal{A} ; \mathcal{A} \mid L :: \bigsqcup_{i \in I} {}_t\, g_i \Rightarrow \mathcal{A}_i \mid \mu X_n \bullet \mathcal{A} \\
\mathcal{S} \ ::= \ & w : \Phi \mid (A_1 \mid \ldots \mid A_n) \mid \phi.
\end{aligned}$$

In order to model scheduling and resource requirements, we extended the TAM Logic Language. We allow our predicates for sequential agents to include a distinguished free variable, R_{req}, representing the resources required by the agent. We also added 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. For specifications and parallel compositions we added another free variable, R_{av}, representing the resources available to the system. A parallel composition may fail if (1) any component fails, (2) there are insufficient resources available, or (3) two components try to use the same resource simultaneously.

We defined a semantic function $\mathcal{F}$ giving the representation of an agent in this model:

$$\mathcal{F} : System \to TAMLL.$$

The semantic definitions satisfy the following condition, namely that agents only use resources during their execution time:[4]

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

In [Low94] we proved a theorem that allows us to split the development of a system into two stages:

[4] This is a slight change from the earlier paper, where we had $\mathcal{F}[\![\mathcal{A} \mid \phi]\!] \Rightarrow ((t, r) \in R_{req} \Rightarrow t \in (t_\alpha, t_\omega])$; our experiences in doing the case study in this paper suggested that this change would make the calculations easier.

- 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.

3 Problem description

The example of this paper is based upon that in [Jac90], and aims to model part of the start-up routine for an aircraft engine. The system under consideration is shown in figure 1. We aim to specify and develop a controller which sits

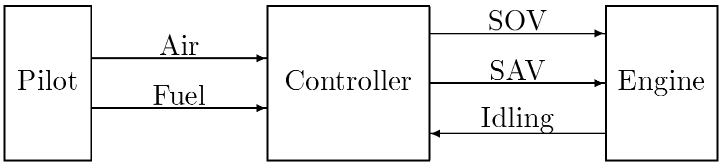

Figure 1: The simplified aircraft engine start-up system

between the pilot and the engine. The pilot may request air or fuel, or request for these to be turned off. 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. The conditions are:

R1: 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;

R2: The shut-off valve should be opened if the pilot requires fuel, as long as the starter air valve is already open;

R3: The starter air valve should be closed when the pilot requests the air to be turned off;

R4: Once the engine reaches idling conditions, the starter air valve should be closed automatically;

R5: Otherwise, if the pilot requests air, the starter air valve should be opened.

We 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. Accordingly, we will make an assumption that the values held in input shunts do not change too quickly, so the delay in sampling them will not be important. The customer must agree to meet this assumption in order to be sure that our program meets his requirements. We will formalize this assumption below.

Formal specification

Our 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.

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

We introduce two specification macros which will prove useful. The first specifies that a shunt s stably holds a particular value x throughout some interval $[t, t')$:

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

The second specifies that value x is written to shunt s within interval $[t, t')$:

$$\mathit{write}(s, x, t, t') \mathrel{\widehat{=}} \exists\, t'' \in [t, t') \bullet s@t'' = (t'', x).$$

We also will want to be able to specify that a shunt write occurs at a particular time t:

$$\mathit{write}(s, x, t) \mathrel{\widehat{=}} s@t = (t, x).$$

As explained above, we need to assume that the values held in input shunts do not change too quickly; we assume that if a value is written to an input shunt, then the shunt remains stable for the following t_{stable} seconds:

$$\begin{aligned} \mathit{ASS_1} \mathrel{\widehat{=}} \forall\, s \in (\mathit{fuel}, \mathit{air}, \mathit{idling}) \bullet \forall\, t \in \mathit{Time} \bullet \\ \mathit{write}(s, x, t) \Rightarrow \mathit{stable}(s, x, t, t + t_{stable}). \end{aligned}$$

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

Similarly, we will specify that the appropriate valves are opened or closed within t_{delay} seconds. Again, we will calculate a value for t_{delay} when we derive our system, and the customer must decide whether this value is acceptable. The reader may safely assume $t_{delay} > t_{stable}$.

Later, when we consider the resource requirements of the system, we will add a further assumption about the resources available.

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:

$$R_1 \mathrel{\widehat{=}} \forall\, t \in [0, T] \bullet \begin{pmatrix} SOV.v@t = open \\ \wedge\ write(fuel, off, t) \end{pmatrix} \Rightarrow write(SOV, close, t, t + t_{delay}).$$

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_{delay} seconds. The assumption ASS_1 tells us that if he asks for fuel at time t, he will not change his mind within t_{stable} seconds.

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

$$R_2 \mathrel{\widehat{=}} \forall\, t \in [0, T] \bullet \begin{pmatrix} SOV.v@t = closed \\ \wedge\ stable(SAV, open, t, t + t_{stable}) \\ \wedge\ write(fuel, on, t) \end{pmatrix} \\ \Rightarrow write(SOV, open, t, t + t_{delay}).$$

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

$$R_3 \mathrel{\widehat{=}} \forall\, t \in [0, T] \bullet \begin{pmatrix} SAV.v@t = open \\ \wedge\ write(air, off, t) \end{pmatrix} \Rightarrow write(SAV, close, t, t + t_{delay}).$$

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

$$R_4 \mathrel{\widehat{=}} \forall\, t \in [0, T] \bullet \begin{pmatrix} SAV.v@t = open \\ \wedge\ write(idling, true, t) \end{pmatrix} \Rightarrow write(SAV, close, t, t + t_{delay}).$$

Otherwise, if the pilot requests air, the starter air valve should be opened:

$$R_5 \mathrel{\widehat{=}} \forall\, t \in [0, T] \bullet \begin{pmatrix} SAV.v@t = closed \\ \wedge\ stable(idling, false, t, t + t_{stable}) \\ \wedge\ write(air, on, t) \end{pmatrix} \\ \Rightarrow write(SAV, open, t, t + t_{delay}).$$

Note that here we also require that the engine does not reach idling conditions shortly after the pilot requests air.

Our complete specification is therefore:

$$\{SAV, SOV\} : ASS_1 \Rightarrow (R_1 \wedge R_2 \wedge R_3 \wedge R_4 \wedge R_5).$$

4 First solution

In this section we derive a system that satisfies our specification. We first derive an initial design, and then calculate a correct scheduler.

Initial design

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

$$\{SOV\} : ASS_1 \Rightarrow R_1 \wedge R_2 \;\mid\; \{SAV\} : ASS_1 \Rightarrow R_3 \wedge R_4 \wedge R_5.$$

We continue with the left hand side:

$$
\begin{array}{ll}
 & \{SOV\} : ASS_1 \Rightarrow R_1 \wedge R_2 \\
\sqsubseteq & \left\langle \text{strengthen specification (rule A.1)} \right\rangle \\
 & \{SOV\} : \forall\, t \in [0, T] \bullet \\
 & \qquad \begin{pmatrix} SOV.v@t = open \\ \wedge\ stable(fuel, off, t, t + t_{stable}) \end{pmatrix} \Rightarrow write(SOV, close, t, t + t_{delay}) \\
 & \qquad \wedge \\
 & \qquad \begin{pmatrix} SOV.v@t = closed \\ \wedge\ stable(SAV, open, t, t + t_{stable}) \\ \wedge\ stable(fuel, on, t, t + t_{stable}) \end{pmatrix} \Rightarrow write(SOV, open, t, t + t_{delay}) \\
\sqsubseteq & \left\langle \text{strengthen specification: } SOV \text{ is either stable or it is written to} \right\rangle \\
 & \{SOV\} : \forall\, t \in [0, T] \bullet \\
 & \qquad \begin{pmatrix} stable(SOV, open, t, t + t_{stable}) \\ \wedge\ stable(fuel, off, t, t + t_{stable}) \end{pmatrix} \Rightarrow write(SOV, close, t, t + t_{delay}) \\
 & \qquad \wedge \\
 & \qquad \begin{pmatrix} stable(SOV, closed, t, t + t_{stable}) \\ \wedge\ stable(SAV, open, t, t + t_{stable}) \\ \wedge\ stable(fuel, on, t, t + t_{stable}) \end{pmatrix} \Rightarrow write(SOV, open, t, t + t_{delay}) \\
\sqsubseteq & \left\langle \begin{array}{l} \text{strengthen specification, choosing } p,\ t'_{stable},\ t'_{delay},\ n \text{ such that} \\ \text{if } k = \left\lceil \frac{t}{p} \right\rceil, \text{ then } [k.p, k.p + t'_{stable}) \subseteq [t, t + t_{stable}) \text{ and} \\ [k.p, k.p + t'_{delay}) \subseteq [t, t + t_{delay}) \text{ and } (n - 1).p \geqslant T \end{array} \right\rangle \\
 & \{SOV\} : \forall\, k \in [0, n - 1] \bullet \\
 & \qquad \begin{pmatrix} stable(SOV, open, k.p, k.p + t'_{stable}) \\ \wedge\ stable(fuel, off, k.p, k.p + t'_{stable}) \end{pmatrix} \Rightarrow \\
 & \qquad\qquad write(SOV, close, k.p, k.p + t'_{delay}) \\
 & \qquad \wedge \\
 & \qquad \begin{pmatrix} stable(SOV, closed, k.p, k.p + t'_{stable}) \\ \wedge\ stable(SAV, open, k.p, k.p + t'_{stable}) \\ \wedge\ stable(fuel, on, k.p, k.p + t'_{stable}) \end{pmatrix} \Rightarrow \\
 & \qquad\qquad write(SOV, open, k.p, k.p + t'_{delay}) \\
\sqsubseteq & \left\langle \text{introduce loop with period p (rule A.3)} \right\rangle
\end{array}
$$

$$
\mathit{Period}\left(p, n, \{SOV\} : \left(\begin{array}{l} \left(\begin{array}{l} \mathit{stable}(SOV, \mathit{open}, t_\alpha, t_\alpha + t'_{stable}) \\ \wedge\ \mathit{stable}(\mathit{fuel}, \mathit{off}, t_\alpha, t_\alpha + t'_{stable}) \end{array}\right) \Rightarrow \\ \qquad \mathit{write}(SOV, \mathit{close}, t_\alpha, t'_{delay}) \\ \wedge \\ \left(\begin{array}{l} \mathit{stable}(SOV, \mathit{closed}, t_\alpha, t_\alpha + t'_{stable}) \\ \wedge\ \mathit{stable}(SAV, \mathit{open}, t_\alpha, t_\alpha + t'_{stable}) \\ \wedge\ \mathit{stable}(\mathit{fuel}, \mathit{on}, t_\alpha, t_\alpha + t'_{stable}) \end{array}\right) \Rightarrow \\ \qquad \mathit{write}(SOV, \mathit{open}, t_\alpha, t'_{delay}) \end{array} \right) \right)
$$

$$
\sqsubseteq \left\langle \begin{array}{l} \text{introduce alternation (rule A.4); strengthen specification;} \\ \text{take } t''_{delay} = t'_{delay} - t'_{stable} \text{ and assume } t''_{delay} > 0 \end{array} \right\rangle
$$

$$
\mathit{Period}\left(p, n, \left(\begin{array}{l} SOV.v = \mathit{open} \wedge \mathit{fuel}.v = \mathit{full} \Rightarrow \\ \qquad \{SOV\} : \mathit{write}(SOV, \mathit{close}, t_\alpha, t_\alpha + t''_{delay}) \\ \sqcup_{t'_{stable}} \\ SOV.v = \mathit{closed} \wedge SAV.v = \mathit{open} \wedge \mathit{fuel}.v = \mathit{on} \Rightarrow \\ \qquad \{SOV\} : \mathit{write}(SOV, \mathit{open}, t_\alpha, t_\alpha + t''_{delay}) \end{array} \right) \right)
$$

$$
\sqsubseteq \left\langle \text{definition of shunt write (rule A.6), deadline (rule A.5)} \right\rangle
$$

$$
\mathit{Period}\left(p, n, \left(\begin{array}{l} SOV.v = \mathit{open} \wedge \mathit{fuel}.v = \mathit{full} \Rightarrow \\ \qquad [t''_{delay}]\mathit{close} \to SOV \\ \sqcup_{t'_{stable}} \\ SOV.v = \mathit{closed} \wedge SAV.v = \mathit{open} \wedge \mathit{fuel}.v = \mathit{on} \Rightarrow \\ \qquad [t''_{delay}]\mathit{open} \to SOV \end{array} \right) \right).
$$

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

$$
\begin{array}{rcl} t'_{stable} + p - 1 & \leqslant & t_{stable} \\ t'_{delay} + p - 1 & \leqslant & t_{delay} \\ t''_{delay} & = & t'_{delay} - t'_{stable} \ > \ 0 \\ p.(n-1) & \geqslant & T. \end{array}
$$

A similar refinement process can be applied to the specification

$$
\{SAV\} : ASS_1 \Rightarrow R_3 \wedge R_4 \wedge R_5,
$$

to obtain the program

$$
\begin{array}{l}
\mathit{Period}\left(p, n, \left(\begin{array}{l} SOV.v = \mathit{open} \wedge \mathit{fuel}.v = \mathit{off} \Rightarrow \\ \qquad [t''_{delay}]\mathit{close} \to SOV \\ \sqcup_{t'_{stable}} \\ SOV.v = \mathit{closed} \wedge SAV.v = \mathit{open} \wedge \mathit{fuel}.v = \mathit{on} \Rightarrow \\ \qquad [t''_{delay}]\mathit{open} \to SOV \end{array} \right) \right) \\
| \\
\mathit{Period}\left(p, n, \left(\begin{array}{l} SAV.v = \mathit{open} \wedge (\mathit{air}.v = \mathit{off} \vee \mathit{idling}.v = \mathit{true}) \Rightarrow \\ \qquad [t''_{delay}]\mathit{close} \to SAV \\ \sqcup_{t'_{stable}} \\ SAV.v = \mathit{closed} \wedge \mathit{idling}.v = \mathit{false} \wedge \mathit{air}.v = \mathit{on} \Rightarrow \\ \qquad [t''_{delay}]\mathit{open} \to SAV \end{array} \right) \right).
\end{array}
$$

Scheduling

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

$$ASS_2 \mathrel{\widehat{=}} R_{av} \supseteq \mathbb{N} \times \{r\}.$$

In order to obtain a scheduler for the program, we label the program constructs requiring scheduling decisions, namely the evaluation of the alternation guards, and the shunt writes. We obtain the program $P_1 \mid P_2$ where

$$P_1 \mathrel{\widehat{=}} Period\left(p, n, L_1 :: \left(\begin{array}{l} SOV.v = open \wedge fuel.v = \mathit{off} \Rightarrow \\ \qquad [t''_{delay}]\ L'_1 :: close \to SOV \\ \bigsqcup_{t'_{stable}} \\ \left(\begin{array}{l} SOV.v = closed \wedge SAV.v = open \\ \wedge\, fuel.v = on \end{array}\right) \Rightarrow \\ \qquad [t''_{delay}]\ L''_1 :: open \to SOV \end{array}\right)\right)$$

$$P_2 \mathrel{\widehat{=}} Period\left(p, n, L_2 :: \left(\begin{array}{l} \left(\begin{array}{l} SAV.v = open \\ \wedge\, (air.v = \mathit{off} \vee idling.v = true) \end{array}\right) \Rightarrow \\ \qquad [t''_{delay}]\ L'_2 :: close \to SAV \\ \bigsqcup_{t'_{stable}} \\ \left(\begin{array}{l} SAV.v = closed \wedge idling.v = false \\ \wedge\, air.v = on \end{array}\right) \Rightarrow \\ \qquad [t''_{delay}]\ L''_2 :: open \to SAV \end{array}\right)\right).$$

Designing a scheduler then amounts to deciding when to perform each of the labelled constructs.

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 seconds; and that it can perform the shunt writes (labelled by L'_1, L''_1, L'_2 and L''_2) within one second each. Thus, 8 seconds of CPU-time will be used in each complete cycle, and so we take $p = 8$. We will attempt to define a scheduler ϕ such that during each 8-second cycle, the guards are evaluated during the first 6 seconds, and the shunt writes are performed during the final 2 seconds. Thus we can calculate values for all the time parameters:

$$\begin{array}{rclrclrcl} p & = & 8 & t'_{stable} & = & 6 & t''_{delay} & = & 2 \\ t_{stable} & = & 13 & t'_{delay} & = & 8 & t_{delay} & = & 15. \end{array}$$

In fact, our system will also meet the specification for any larger values of the time parameters t_{stable} and t_{delay}.

More definitely, our scheduler ϕ will ensure that P_1's guards are evaluated during the first three seconds, then P_2's guards are evaluated during the next three seconds, then P_1 performs its shunt write, and finally P_2 performs its shunt write. We will try to ensure:

$$\begin{array}{rcl} \mathcal{F}[\![P_1 \mid \phi]\!] & \Rightarrow & \neg\, \mathsf{fail} \wedge R_{req} \subseteq \{(t, r) \mid t \bmod 8 \in \{0, 1, 2, 6\}\} \\ \mathcal{F}[\![P_2 \mid \phi]\!] & \Rightarrow & \neg\, \mathsf{fail} \wedge R_{req} \subseteq \{(t, r) \mid t \bmod 8 \in \{3, 4, 5, 7\}\}. \end{array}$$

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 $8m$, and so the labels L_1 and L_2 are encountered at times $8m$ (for $m \in [0, n)$), and labels L'_1, L''_1, L'_2 and L''_2 are encountered at times $8m+6$ (for $m \in [0, n)$). Thus we will schedule P_1's alternation immediately and delay P_2's for 3 seconds:

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

And we will schedule P_1's shunt writes immediately, and delay P_2's for one second:

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

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

5 An alternative derivation

In the above derivation we produced a system which met all requests within $t_{delay} = 15$ seconds assuming the inputs remain stable for $t_{stable} = 13$ seconds. In this section we try to produce a system that works for smaller values of these constants.

The key to achieving this stronger specification will be to consider the program derivation and scheduling together, unlike in the previous section where we considered these separately. Our specification is as before, except we assume both ASS_1 and ASS_2, and we add the extra condition that the system does not fail:

$$\{SAV, SOV\} : ASS_1 \wedge ASS_2 \Rightarrow R_1 \wedge R_2 \wedge R_3 \wedge R_4 \wedge R_5 \wedge \neg\, \mathsf{fail}.$$

As before, we refine this to a parallel composition, and each side will be implemented as an 8 second cycle. However, this time we will specify that in each 8 second cycle, the left hand side uses the CPU for the first 4 seconds and the right hand side uses the CPU for the next 4 seconds:

$$\begin{array}{l} \{SOV\} : ASS_1 \Rightarrow \begin{pmatrix} R_1 \wedge R_2 \wedge \neg\, \mathsf{fail} \\ \wedge\, R_{req} \subseteq \{(t, r) \mid t \bmod 8 \in \{0, 1, 2, 3\}\} \end{pmatrix} \\ | \\ \{SAV\} : ASS_2 \Rightarrow \begin{pmatrix} R_3 \wedge R_4 \wedge R_5 \wedge \neg\, \mathsf{fail} \\ \wedge\, R_{req} \subseteq \{(t, r) \mid t \bmod 8 \in \{4, 5, 6, 7\}\} \end{pmatrix}. \end{array}$$

This refinement can be justified by ASS_2, which says that all the resources required by the two sides are available.

We continue with the left hand side:

$$\{SOV\} : ASS_1 \Rightarrow \begin{pmatrix} R_1 \wedge R_2 \wedge \neg\, \mathsf{fail} \\ \wedge\, R_{req} \subseteq \{(t, r) \mid t \bmod 8 \in \{0, 1, 2, 3\}\} \end{pmatrix}$$

$$\sqsubseteq \left\langle \text{strengthen specification, as above} \right\rangle$$

$$\{SOV\} : \forall\, t \in [0, T] \bullet$$
$$\begin{pmatrix} stable(SOV, open, t, t + t_{stable}) \\ \wedge\, stable(fuel, off, t, t + t_{stable}) \end{pmatrix} \Rightarrow write(SOV, close, t, t + t_{delay})$$
$$\wedge$$
$$\begin{pmatrix} stable(SOV, closed, t, t + t_{stable}) \\ \wedge\, stable(SAV, open, t, t + t_{stable}) \\ \wedge\, stable(fuel, on, t, t + t_{stable}) \end{pmatrix} \Rightarrow write(SOV, open, t, t + t_{delay})$$
$$\wedge\, \neg\, \mathsf{fail} \wedge R_{req} \subseteq \{(t, r) \mid t \bmod 8 \in \{0, 1, 2, 3\}\}$$

$$\sqsubseteq \left\langle \begin{array}{l} \text{strengthen specification, choosing } t'_{stable},\ t'_{delay},\ n \text{ such that} \\ \text{if } k = \lceil \frac{t}{8} \rceil, \text{ then } [8k, 8k + t'_{stable}) \subseteq [t, t + t_{stable}) \\ \text{and } [8k, 8k + t'_{delay}) \subseteq [t, t + t_{delay}) \text{ and } 8(n-1) \geqslant T \end{array} \right\rangle$$

$$\{SOV\} : \forall\, k \in [0, n-1] \bullet$$
$$\begin{pmatrix} stable(SOV, open, 8k, 8k + t'_{stable}) \\ \wedge\, stable(fuel, off, 8k, 8k + t'_{stable}) \end{pmatrix} \Rightarrow$$
$$write(SOV, close, 8k, 8k + t'_{delay})$$
$$\wedge$$
$$\begin{pmatrix} stable(SOV, closed, 8k, 8k + t'_{stable}) \\ \wedge\, stable(SAV, open, 8k, 8k + t'_{stable}) \\ \wedge\, stable(fuel, on, 8k, 8k + t'_{stable}) \end{pmatrix} \Rightarrow$$
$$write(SOV, open, 8k, 8k + t'_{delay})$$
$$\wedge\, \neg\, \mathsf{fail} \wedge R_{req} \subseteq \{(t, r) \mid t \bmod 8 \in \{0, 1, 2, 3\}\}$$

$$\sqsubseteq \left\langle \text{introduce loop with period 8} \right\rangle$$

$$Period \left(8, n, \left(\begin{array}{l} \begin{pmatrix} stable(SOV, open, t_\alpha, t_\alpha + t'_{stable}) \\ \wedge\, stable(fuel, off, t_\alpha, t_\alpha + t'_{stable}) \end{pmatrix} \Rightarrow \\ \qquad write(SOV, close, t_\alpha, t_\alpha + t'_{delay}) \\ \wedge \\ \begin{pmatrix} stable(SOV, closed, t_\alpha, t_\alpha + t'_{stable}) \\ \wedge\, stable(SAV, open, t_\alpha, t_\alpha + t'_{stable}) \\ \wedge\, stable(fuel, on, t_\alpha, t_\alpha + t'_{stable}) \end{pmatrix} \Rightarrow \\ \qquad write(SOV, open, t_\alpha, t_\alpha + t'_{delay}) \\ \wedge\, \neg\, \mathsf{fail} \wedge R_{req} \subseteq \{t_\alpha, t_\alpha + 1, t_\alpha + 2, t_\alpha + 3\} \times \{r\} \end{array} \right) \right)$$

$$\sqsubseteq \left\langle \begin{array}{l} \text{introduce alternation; strengthen specification;} \\ \text{take } t''_{delay} = t'_{delay} - t'_{stable} > 0 \end{array} \right\rangle$$

$$
Period\left(8, n, L_1 :: \left(\begin{array}{l}
SOV.v = open \wedge fuel.v = full \Rightarrow \\
\qquad \{SOV\} : \left(\begin{array}{l} write(SOV, close, t_\alpha, t_\alpha + t''_{delay}) \\ \wedge \neg \mathsf{fail} \\ \wedge\ R_{req} \subseteq [t_\alpha, t_\alpha + t''_{delay}) \times \{r\} \end{array}\right) \\
\bigsqcup_{t'_{stable}} \\
SOV.v = closed \wedge SAV.v = open \wedge fuel.v = on \Rightarrow \\
\qquad \{SOV\} : \left(\begin{array}{l} write(SOV, open, t_\alpha, t_\alpha + t''_{delay}) \\ \wedge \neg \mathsf{fail} \\ \wedge\ R_{req} \subseteq [t_\alpha, t_\alpha + t''_{delay}) \times \{r\} \end{array}\right)
\end{array}\right)\right)
$$

$| \ \phi.$

We want to minimise the values of the time parameters, so we will schedule the guard evaluation immediately:

$$\phi(t, L_1) = \{(t, r), (t + 1, r), (t + 2, r)\}.$$

Thus we can take

$$t'_{stable} = 3 \qquad t_{stable} = 10.$$

Using the definition of shunt write, we have:

$$
\begin{array}{l}
\{SOV\} : \left(\begin{array}{l} write(SOV, close, t_\alpha, t_\alpha + t''_{delay}) \\ \wedge \neg \mathsf{fail} \\ \wedge\ R_{req} \subseteq [t_\alpha, t_\alpha + t''_{delay}) \times \{r\} \end{array}\right) \ \sqsubseteq \ L'_1 :: close \to SOV \mid \phi \\
\{SOV\} : \left(\begin{array}{l} write(SOV, open, t_\alpha, t_\alpha + t''_{delay}) \\ \wedge \neg \mathsf{fail} \\ \wedge\ R_{req} \subseteq [t_\alpha, t_\alpha + t''_{delay}) \times \{r\} \end{array}\right) \ \sqsubseteq \ L''_1 :: open \to SOV \mid \phi,
\end{array}
$$

where the shunt writes are scheduled immediately:

$$\phi(t, L'_1) = \{(t, r)\} \qquad \phi(t, L''_1) = \{(t, r)\}.$$

Again we are seeking to minimise the time parameters, so we can take

$$t''_{delay} = 1 \qquad t'_{delay} = 4 \qquad t_{delay} = 11.$$

A similar refinement process can be applied to the right hand side of the parallel composition, but here the operations have to be delayed for 4 seconds. Putting the results together, we have the system $P_1 \mid P_2 \mid \phi$ where

$$
P_1 \ \widehat{=} \ Period\left(8, n, L_1 :: \left(\begin{array}{l}
SOV.v = open \wedge fuel.v = off \Rightarrow \\
\qquad L'_1 :: close \to SOV \\
\bigsqcup_3 \\
\left(\begin{array}{l} SOV.v = closed \wedge SAV.v = open \\ \wedge\ fuel.v = on \end{array}\right) \Rightarrow \\
\qquad L''_1 :: open \to SOV
\end{array}\right)\right)
$$

$$P_2 \;\widehat{=}\; Period\left(8, n, \Delta 4;\; L_2 :: \left(\begin{array}{l} \left(\begin{array}{l} SAV.v = open \\ \wedge\ (air.v = off \vee idling.v = true) \end{array} \right) \Rightarrow \\ \qquad L_2' :: close \rightarrow SAV \\ \sqcup_3 \\ \left(\begin{array}{l} SAV.v = closed \wedge idling.v = false \\ \wedge\ air.v = on \end{array} \right) \Rightarrow \\ \qquad L_2'' :: open \rightarrow SAV \end{array} \right) \right),$$

and

$$\begin{array}{rcl} \phi(t, L_1) & = & \{(t, r), (t+1, r), (t+2, r)\} \\ \phi(t, L_2) & = & \{(t, r), (t+1, r), (t+2, r)\} \end{array}$$

$$\begin{array}{rclcrcl} \phi(t, L_1') & = & \{(t, r)\} & \qquad & \phi(t, L_1'') & = & \{(t, r)\} \\ \phi(t, L_2') & = & \{(t, r)\} & \qquad & \phi(t, L_2'') & = & \{(t, r)\}. \end{array}$$

6 Conclusions

In this paper we have applied the scheduling-oriented model for TAM to an example concerning part of the start-up routine for an aircraft engine. We used the TAM logic language to specify the system. We used the TAM refinement rules to produce an initial design, and then used the scheduling model to define and verify a scheduler. In section 5 we adopted a different approach, considering the program and scheduler together, which enabled us to produce a solution meeting tighter deadlines.

We have demonstrated that the models can be used in two different ways, either splitting the development process into two stages, or considering both the program and scheduler simultaneously. The latter approach will, generally, produce better solutions, but at the price of a harder derivation. We believe that the former approach will normally be preferable unless the deadlines are particularly tight; even if this approach does not produce a solution, it will help to identify bottle-necks, and so when the second approach is used, the programmer knows where the problems lie, and so can design the whole system with this in mind.

The case study has also revealed a couple of places where the original semantic model needed changing, noted in the footnotes on pages 3 and 4.

We would like to undertake further case studies using the scheduling-oriented model. We believe that this will help us to develop refinement-style rules for deriving schedulers. One such rule, which we implicitly used at the beginning of section 5, is:

If $w \cap w' = \emptyset$ and $R \cap R' = \emptyset$ and $ASS \Rightarrow R \cup R' \subseteq R_{av}$ then

$$w \cup w' : ASS \Rightarrow (\Phi \wedge \Phi' \wedge \neg\, \mathsf{fail}) \;\sqsubseteq$$

$$w : ASS \Rightarrow \left(\begin{array}{l} \Phi \wedge \neg\, \mathsf{fail} \\ \wedge\ R_{req} \subseteq R \end{array} \right) \;\Big|\; w' : ASS \Rightarrow \left(\begin{array}{l} \Phi' \wedge \neg\, \mathsf{fail} \\ \wedge\ R_{req} \subseteq R' \end{array} \right).$$

To show that a parallel composition does not fail it is enough to show that neither component fails, that they use disjoint resources, and that neither uses resources that are not available.

Acknowledgements

I would like to thank He Jifeng and Tony Hoare for useful comments and suggestions on this paper.

References

[DRRS94] J. Davies, G. M. Reed, A. W. Roscoe, and S. A. Schneider. *Real Time CSP*. Prentice Hall, 1994. Forthcoming.

[DS94] Jim Davies and Steve Schneider. A brief history of timed CSP. *Theoretical Computer Science*, 1994. Forthcoming.

[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. Scheduling-oriented models for real-time systems. Technical Report PRG-TR-10-94, Oxford University Computing Laboratory, 1994.

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

[Rol88] Rolls-Royce plc. *RB211-524G & H DFC System Requirements for Autostart, Autoignition and Autorelight*, 1988. Technical Design Report TDR9456.

[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.

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

Rule A.2 (Parallel composition): If $w \cap w' = \emptyset$, then

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

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

$$w : \left(\forall\, k \in [0, n) \bullet \Phi[p.k/t_\alpha] \right) \;\sqsubseteq\; \mathit{Period}(p, n, w : \Phi).$$

Rule A.4 (Alternation): If

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

then

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

Rule A.5 (Deadline): $w : \Phi \wedge t_\omega - t_\alpha = t \;\sqsubseteq\; [t](w : \Phi)$.

Rule A.6 (Shunt write): If $s \in w$ then $w : \mathit{write}(s, x, t_\alpha, t_\omega) \sqsubseteq x \rightarrow s$.