ON THE DESIGN OF SIDE-STICK CONTROLLERS IN FLY-BY-WIRE AIRCRAFT

Muffy Thomas and Bowen Ormsby*
University of Glasgow, Scotland, U.K.

Abstract

This paper presents the problem of designing the functional behaviour of the interaction between two side-stick controllers, an autopilot, and a flight control computer in a fly-by-wire aircraft. Two models are developed using the ISO formal description technique LOTOS, and analysed using rigorous abstract testing techniques.

Keywords: Avionics, Design, Formal Description Techniques, LOTOS, Process Algebras, Safety-Critical Software, Specification, Testing, Verification, Validation.

Introduction

The reliability of software in embedded computer systems is a matter of increasing concern, particularly for safety-critical systems.

There are no known quantitative methods for demonstrating high levels of software reliability [7, 8]; instead, most approaches to software in safety-critical situations rely mainly on *assurance*, based on the evidence produced during the software lifecycle. Our interest lies in augmenting the preliminary and detailed design stages with mathematical modelling and validation stages; our goal is the reduction of errors and design changes during unit testing *after* implementation (i.e. by finding the errors *before* implementation). Thus, we are interested in applying formal methods as *design* tools.

The aerospace industry is an area where software, on the whole, is extending existing electromechanical systems; the concerns of control and protection must be clearly separated and qualified, with a high degree of assurance. It is becoming clear from our own experience, from others in the field [10, 11, 1], and in the aerospace industry (eg. [3]), that techniques such as static analysis and validation based on mathematical models can have an important role to play in the design of safety-critical software. This is particularly the case when analysis addresses the questions of whether the software satisfies safety constraints.

Here, we consider a case study: the problem of designing the functional behaviour of the interactions between two side-stick controllers, the autopilot, and a flight control computer in a fly-by-wire aircraft. This is a relevant example since such a configuration is part of the current Airbus Industrie A320 [4]. Our aim is to develop a model from an initial statement of the requirements, to verify that the model does fulfil the requirements, and to provide some degree of assurance that this is a good design by analysing the model with respect to some safety requirements. Two models, of increasing complexity, are developed and analysed using the ISO formal description technique LOTOS (*Language of Temporal Ordering Specification*) [5]. LOTOS is appropriate since we are concerned only with relative time, rather than with real time.

*Funded by University of Glasgow Research Studentship

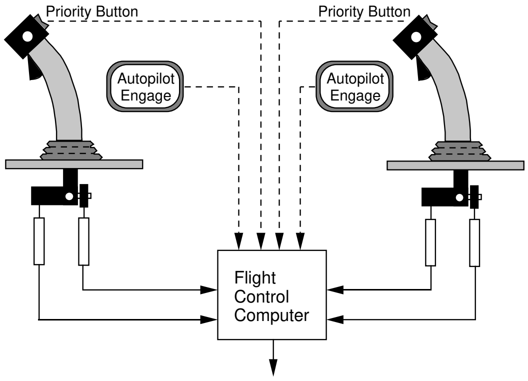

Figure 1: Side-Stick Controllers and FCC

The first section gives the informal requirements of the side-sticks controllers, and the second section introduces LOTOS. The basic design description is given in the third section, followed by an analysis of the design. Then, the design is extended to include time, and this too is analysed. There follows a discussion and our conclusions.

1 Requirements

The requirements have been extracted from [4].

Each pilot has a side-stick, and the two sticks are linked to the flight control computer as shown in figure 1. Normally, both sticks are enabled and movements of them produce a response which depends on the sum of their displacements, up to a maximum of the full deflection of one stick. So, if both sticks are moved forward two degrees, the aircraft will behave as if one stick had been moved forward four degrees. Similarly, equal and opposite movements cancel each other out.

In an emergency, one pilot can override the other pilot's controls by pressing a button on his/her stick. This button (which also acts as the autopilot disengage button) causes inputs from the other pilot's stick to be ignored, and gives priority control to the pilot who pressed it. If the button is held down for more than thirty seconds, the stick retains priority until the system is reset by centralising the disengaged stick. This override system allows one pilot to take sole control of the aircraft if the other pilot gets into difficulty or if a stick jams.

Amongst many other controls, both pilots also have an autopilot engage button. If both pilots are in control (i.e. not overridden) and one of them presses this button, the autopilot also gains control of the aircraft until one of the pilots presses the autopilot disengage button on his/her side-stick. A pilot in control can engage or disengage the autopilot, but a pilot not in control cannot affect the state of the autopilot.

2 Design Descriptions

The natural language description given above, whilst relatively clear, is still ambiguous and incomplete. Further, it offers no possibility of rigorously demonstrating that it is a good, safe protocol that can be implemented.

Our intention is to develop a design description which is closer to an eventual implementation, but abstract enough to allow for rigorous analysis of the essential behaviour.

The formal description technique LOTOS is used to model the design; LOTOS is a process algebra (similar to CCS [9] but with data types and multi-way synchronisation) which allows us to model processes that can be non-deterministic and/or concurrent, with or without synchronisation. Processes can be parameterised by abstract data type values, and such values can be passed, or negotiated, between processes through synchronisation.

A good introduction to LOTOS is given in [2]; a very brief review of the features of LOTOS we use is as follows. Processes are built up from constant processes, events, and process operators. Events are atomic, indivisible actions. In the following, `P` and `Q` are processes.

LOTOS	*Informal description*
`exit`	termination
`a;P`	prefix `P` by event `a`
`P [] Q`	choice between `P` and `Q`
`P >> Q`	become `Q` after `P` terminates
`P ||| Q`	`P` in parallel with `Q`
`P |[l]| Q`	`P` in parallel with `Q`, synchronising on events in list `l`
`[exp] -> P`	if `exp` holds then become `P`

Abstract data types are specified (in the ACT ONE sublanguage) using many-sorted equational logic. Values and processes are combined in two ways: lists of values may be associated with events, and processes may be parameterised by values. We use two forms of value association with events: the event offer `a!v` offers value `v` at event `a`, and the event offer `a?v:T` offers any value of type T at event `a`. Two events offering equivalent values may synchnronise, e.g. `a!true` can synchronise with `a!true`, or `a!not(false)`, and an event offer `a!v1` can synchronise with an event offer `a?v2:T`, with the effect that `v2` is bound to `v1`, when `v1` has type `T`.

The underlying semantics of a LOTOS description is a state transition system, rather like a finite state automata, except that there is no restriction to finite states. We use laws which describe equivalences between processes, based on the concept of weak bisimulation [9] between transition systems. (Bisimulation is so-called because the processes can simulate each other.) These laws include, for example, the rule that choice is commutative, and that parallel processes expand into processes involving only choice and prefix.

For brevity, all the events in our descriptions will be observable. This will allow us to omit the usual list of observable, parameter events in process declarations, except in the few cases where the actual and formal events do indeed differ. We also introduce some further, trivial, abuses of LOTOS notation, for brevity.

3 Basic Design

We begin with a design of the basic system, excluding timing constraints.

Four data types are defined to represent the status of a stick, a pilot's controls (`pcontrols`), the control priority (`priority`) and the `autopilot`. In the interest of brevity, most of the (obvious) equations are omitted from these types.

A stick can be in any position along an axis from n4 (negative 4) to p4 (positive 4), with 00 as origin. Stick positions may be summed, with over and underflows rounded up or down to n4, p4 respectively.

```
type stickpos is boolean
   sorts stickpos
   opns n4,n3,n2,n1,00,p1,p2,p3,p4 :  -> stickpos
      Succ , Pred : stickpos          -> stickpos
      _+_ ,_eq_   : stickpos,stickpos -> stickpos
   eqns forall x,y:stickpos
      ...
endtype
```

A pilot's controls consists of the x and y coordinates of a stick; the coordinates are selected with operations `xstick` and `ystick`, and the coordinates are incremented and decremented with operations `xinc`, `xdec`, etc. The operation `central` is a predicate to test whether both coordinates are at the origin. Again, many of the equations are omitted.

```
type pilotcontrols is stickpos
   sorts pcontrols
   opns mk: stickpos,stickpos    -> pcontrols
        xstick,ystick: pcontrols -> stickpos
        xinc,xdec: pcontrols     -> pcontrols
        yinc,ydec: pcontrols     -> pcontrols
        central: pcontrols       -> bool
   eqns forall x,y:stickpos
     ofsort stickpos
        xstick(mk(x,y)) = x;
        ystick(mk(x,y)) = y;
     ofsort pcontrols
        xinc(mk(x,y)) = mk(succ(x),y);
        xdec(mk(x,y)) = mk(pred(x),y);
        ...
     ofsort bool
        central(mk(x,y))= ((x eq 00) and (y eq 00))
endtype
```

There are 3 distinct values for priority status.

```
type priority is boolean
   sorts priority
   opns one,two,none : -> priority
      _eq_ :priority, priority -> bool
   eqns
      ...
endtype
```

There are 2 values for autopilot status.

```
type autopilot is boolean renamedby
   sortnames autopilot for bool
   opnnames on for true
            off for false
endtype
```

The main process components of the system are the two (human) pilots, referred to as pilot 1 and pilot 2, the autopilot, and the flight control computer.

The two (human) pilots' behaviours are identical, up to the names of the events, and so the process is parameterised. Each pilot may move the stick along one of the four axes: forward (Fd), backward (Bk), left (Lt), right (Rt); depress or release their priority button: (P) and (Pr) respectively, and depress or release their autopilot button: (AU) and (AuD) respectively.

```
process Pilot[Fd,Bk,Lt,Rt,Au,AuD,P,PR]:=
    (Fd [] Bk [] Lt [] Rt []
     Pr [] Au [] P [] AuD);Pilot
 [] exit
endproc
```

The autopilot has a more restricted functionality which includes only stick movements.

```
process AutoPilot[Fd,Bk,Lt,Rt] :=
    (Fd [] Bk [] Lt [] Rt);AutoPilot
 [] exit
endproc
```

The flight control computer polls for the current (summed) positions of the sticks and the priorities with event s, and sends on the appropriate signal with event signal to the control surfaces. There are several possibilities for combining the inputs into the appropriate signal, depending on which (human) pilot (if any) is in overall control, and whether or not the autopilot is engaged.

```
process FCC :=
   s?pi1,pi2,pi3:pcontrols!one!on;
   signal!mk(xstick(pi1)+xstick(pi3),
             ystick(pi1)+ystick(pi3));
   FCC[s,signal]
[] s?pi1,pi2,pi3:pcontrols!one!off;
   signal!mk(xstick(pi1),ystick(pi1));
   FCC[s,signal]
[] s?pi1,pi2,pi3:pcontrols!two!on;
   signal!mk(xstick(pi2)+xstick(pi3),
             ystick(pi2)+ystick(pi3));
   FCC[s,signal]
[] s?pi1,pi2,pi3:pcontrols!two!off;
   signal!mk(xstick(pi2),ystick(pi2));
   FCC[s,signal]
[] s?pi1,pi2,pi3:pcontrols!none!on;
   signal!mk(xstick(pi1)+xstick(pi2)+xstick(pi3),
             ystick(pi1)+ystick(pi2)+ystick(pi3));
   FCC[s,signal]
[] s?pi1,pi2,pi3:pcontrols!none!off;
   signal!mk(xstick(pi1)+xstick(pi2),
             ystick(pi1)+ystick(pi2));
   FCC[s,signal]
[] exit
endproc
```

The effects of the pilots' actions on the FCC are defined by the process State, which is composed, in a *constraint-oriented* style, with the three pilots and the FCC. The constraint-oriented style [13] is one in which constraints are regarded as behavioural properties of the system, and the complete system is formed by combining all of these properties using a form of parallelism. Unconstrained parallelism corresponds to the disjunction of the behavioural properties, and constrained parallelism, i.e. with synchronisation, corresponds to a form of conjunction (depending on the synchronising events).

The constraint-oriented style is the best approach, for this problem, for two reasons. First, in the design process, it allows us to separate concerns and decouple the main components of the system: the pilots and the FCC. We can consider them in isolation, as independent agents, and then define their interaction. Second, this approach best reflects the separation of components in the intended hardware/software implementation.

The overall process is given by the appropriate combination of the five components.

```
process Controller(pi1,pi2,pi3:pcontrols,
                   p:priority,a:autopilot):=
   State(pi1,pi2,pi3,p,a)
      |[all events]|
   (Pilot1 ||| Pilot2 ||| AutoPilot)
      |[s]|
   FCC[s,signals]
endproc
```

where
Pilot1 = Pilot[F1,B1,L1,R1,A1,AD1,P1,PR1],
Pilot2 = Pilot[F2,B2,L2,R2,A2,AD2,P2,PR2], and
AutoPilot = AutoPilot[F3,B3,L3,R3].

The State process is where the behaviour of the interaction between the pilots is defined. All pilot events are possible, i.e. a pilot can always apply a force to his/her stick/button, but that event may have no observable effect. Since much of the details concerning the three pilots is similar, we have omitted parts of this process (unfortunately, LOTOS is not higher order and does not allow parameterisation over processes).

```
process State(pi1,pi2,pi3:pcontrols,
              p:priority,a:autopilot):=
   F1;State(yinc(pi1),pi2,pi3,p,a)
[] F2;State(pi1,yinc(pi2),pi3,p,a)
[] F3;State(pi1,pi2,yinc(pi3),p,a)

...similarly for B1,B2,B3,R1,R2,R3,L1,L2,L3...

[] P1;([not (p eq two)]->State(pi1,pi2,pi3,one,off)
    [] [p eq two]      ->State(pi1,pi2,pi3,p,a))
[] P2;([not (p eq one)]->State(pi1,pi2,pi3,two,off)
    [] [p eq one]      ->State(pi1,pi2,pi3,p,a))
[] PR1;([p eq one]     ->State(pi1,pi2,pi3,none,off)
    []  [not(p eq one)]->State(pi1,pi2,pi3,p,a))
[] PR2;([p eq two]     ->State(pi1,pi2,pi3,none,off)
    []  [not(p eq two)]->State(pi1,pi2,pi3,p,a))
[] A1;([not(p eq two)] ->State(pi1,pi2,pi3,p,on)
    [] [p eq two]      ->State(pi1,pi2,pi3,p,a))
[] A2;([not(p eq one)] ->State(pi1,pi2,pi3,p,on)
    [] [p eq one)]     ->State(pi1,pi2,pi3,p,a))
[] AD1;([not(p eq two)]->State(pi1,pi2,pi3,p,off)
     [] [p eq two]     ->State(pi1,pi2,pi3,p,a))
[] AD2;([not(p eq one)]->State(pi1,pi2,pi3,p,off)
     [] [p eq one]     ->State(pi1,pi2,pi3,p,a))
[] (s!pi1!pi2!p!a;State(pi1,pi2,pi3,p,a))
[] exit
endproc
```

Note that the effect of the priority and autopilot button events depend on the overall pilot priority.

4 Analysis

Our analysis of the design has two aims: to verify that it does fulfill the requirements, and that to show that it is a good design with respect to some basic safety properties.

Some of them are:

1. Moving a pilot's stick cannot affect the state of control nor of the autopilot.
2. If the autopilot is on, then the signal sums the autopilot stick positions with the pilots' stick positions (depending on who is in control).
3. If the autopilot is off, then the autopilot stick positions are not summed in the signal.
4. If both pilots are in control (i.e. neither is overridden) and one of them presses the autopilot button, the autopilot also gains control of the aircraft until one of the pilots presses the autopilot disengage button on their side-stick.
5. A pilot in control can engage or disengage the autopilot, but a pilot not in control cannot affect the state of the autopilot.

Unlike the description of the flight warning computer in LOTOS in [3], we are able to perform rigorous analysis directly on our design. (In [3], "black-box" testing was performed on an ADA implementation.)

Specifically, we perform an abstract form of testing, called property testing which is described in and used extensively in [12]. Testing, in LOTOS, is a form of state reachability analysis; property testing is a more abstract form of testing for a specific property. The property is defined as a LOTOS process, and then the test process is combined, in parallel, synchronising on the events of the test process, with the given process. Essentially, we are testing to see if the given process *can* behave like the test process, without deadlock. If the test is passed, then we can be assured that the property does hold, in that the test behaviour is possible. On the other hand, if the test is not passed, then we can conclude only that the behaviour is not possible for the states considered so far.

This illustrates an important point: there are always more states to explore because there are infinitely many processes, i.e. the processes have the form `P := exit [] ...;P`. We can only reason conclusively, using testing, over a finite number of states. However, if we detect duplicate states, then we can reason about recursive processes, using a fixed point theorem. This means that if we can find a recursive definition of the combined process under inspection, then if the test cannot be passed in the non-recursive prefixes, we can conclude that the test can never be passed.

Property testing is not the most sophisticated verification technique in comparison to, say, a temporal/modal logic. But, it is particularly attractive to us because it can be performed with the software tool LOLA (see [13]) - a simulation tool for symbolic computation that can recognise duplicate states. Also, and no less important, we use LOLA for straightforward prototyping, or animation, when developing the descriptions.

We cannot discuss each property in detail here, but as one example, consider testing for the first property. Because this is a general property, i.e. not one about a specific instance of the controller, the test process depends on the free variables in the given process, say `Controller(x,y,p,a)`. At first, it seems then that the corresponding test process is the process which offers any number of stick events from the three pilots, followed by an offer of the `s` event (the event which passes information to the flight controller) with the priority and autopilot values unchanged, i.e.

```
Controller(x,y,p,a)
   |[all events]|
((Sticks[F1,B1,R1,L1] ||| Sticks[F2,B2,R2,L2]
  ||| Sticks[F3,B3,R3,L3])
 >>(s?pi1,pi2,pi3:pcontrols!p!a;exit))
```

where `Sticks[...]` is just identical to `AutoPilot`. Using LOLA, the test `is` passed, with a suitable depth of state exploration. However, this only shows that it is *possible* that the stick actions do not affect the priority or autopilot status; it does not show that they cannot have such an effect. In order to show this, we need to show that tests such as

```
Controller(x,y,p,on)
   |[all events]|
((Sticks[F1,B1,R1,L1] ||| Sticks[F2,B2,R2,L2]
  ||| Sticks[F3,B3,R3,L3])
 >>(s?pi1,pi2,pi3:pcontrols!p!off;exit))
```

cannot be passed. Using LOLA, we are able to transform this particular process into a (rather large and complicated) set of recursion equations; examination of the non-recursive prefixes reveals that this test cannot be passed.

Demonstrating that the other properties hold involves considerably more complex test processes. It is easier to demonstrate that a refinement of a property holds. For example, we can easily show a refinement of property 4:

```
Controller(x,y,none,off)
   |[all events]|
(P1;
(Sticks[F1,B1,R1,L1] ||| Sticks[F2,B2,R2,L2]
  ||| Sticks[F3,B3,R3,L3])
>>
A2; s?pi1,pi2,pi3:pcontrols!none!off;exit)
```

Informally, we are testing for: given no pilot is in overall control and the autopilot is off, if pilot 1 depresses his/her priority button, and then after any number of events which do not include any button events, pilot 2 depresses his/her autopilot, the autopilot is still off.

One requirement which we are not able to express and verify in this design is one concerning time: namely, the behaviour when a priority button has been engaged for a length of time. In order to express this, we will need to extend the design to include a notion of time; we will do so in Section 5.

4.1 Discussion: the Design Process

The LOTOS model has been invaluable; it has allowed us to precisely define our design, to test it, and analyse it. Moreover, the activity of describing the design in LOTOS made us think very carefully, and hard, about the interactions of the various components of the system. We were often forced to determine the interaction between pilot control and the autopilot, in more detail, than was explicitly discussed in the requirements. For example, when the autopilot is disengaged, what is the control priority? (We conclude that it remains unchanged.)

Also, we did get some aspects of the design clearly wrong. For example, at first, we could not show the fifth property. This was because in the process `State` we had the choices

```
AD1;State(pi1,pi2,p,off) [] AD2;State(pi1,pi2,p,off)
```

i.e. *any* pilot can switch off the autopilot. But, the requirements clearly state that a pilot *not* in control cannot affect the state of the autopilot, and in our design, a pilot is only *not* in control when the other pilot has sole control, e.g. pilot 1 is not in control when the priority is two. Fortunately, through the process of testing, we found the error, and corrected it accordingly. A final lesson learned was: catch design errors *early*, develop the design iteratively, and save the iterations carefully. In other words, careful management of the design process is essential.

5 Timed Design

In this section we discuss how to augment our design with an explicit model of time. We will not deal with real time, since we do not know what these constraints are for this problem. There are various ways to model time; we have chosen to take an approach which is consistent with the interleaving model of parallelism (rather than true concurrency) which underlies LOTOS. All processes synchronise on one clock, with the tick event `t`. The tick is assumed to be of such a short duration that at most one event can occur at each tick, although no event is required to occur.

First, we extend the current description in a constraint-oriented style, with a clock process. In addition to the tick event, this process offers an event `now` which is associated with the current value of the clock.

```
process Clock[t,now](current:Nat):=
   t; Clock[t,now](Succ(current))
[] now!current; Clock[t,now](current)
[] exit
endproc
```

Each high level process in the controller process is augmented with an initial offer of the tick event. So, for example instead of `(Pilot1 ||| Pilot2 ||| AutoPilot)` we define a process `Pilots`.

```
process Pilots :=
   t;  (Pilot1 ||| Pilot2 ||| AutoPilot)
   >> Pilots
endproc
```

Second, we extend the design to model the behaviour when a priority button has been engaged for a length of time, e.g. 30 seconds. We will assume here, without loss of generality, that a tick represents a second. Again, we extend the description in a constraint-oriented style with a process `Button` which stores the last time of depression of a priority button. Event `wb` models the write action and event `rb` the read action.

```
process Button[rb,wb](time:Nat):=
   wb?btime:Nat; Button[rb,wb](btime)
[] rb!time; Button[rb,wb](time)
[] exit
endproc
```

The process `State` is extended to reflect three different modes. These are: as before, i.e. neither pilot has overall control; pilot 1 has what we call *absolute* control; or pilot 2 has *absolute* control. By *absolute* control we mean that a pilot has had sole priority for more than 30 seconds.

These three modes are modelled by subprocesses `State1` (which is essentially the old process `State`, but it recursively calls the new `State` instead of `State1`), `State2one` and `State2two`.

The two latter processes are identical, modulo event and value renaming. `State2one`, for example, defines the behaviour when pilot 1 has absolute control. Since the effects of the stick movements are exactly as in `State1`, these are omitted.

```
process State2one
(pi1,pi2,pi3:pcontrols,p:priority,a:autopilot) :=
 t;
(  ...
[] P1;State2(pi1,pi2,pi3,p,a)
[] P2;[central(pi2)]->(now?time:Nat;wb!time;
                       State(pi1,pi2,pi3,two,a))
  [] [not(central(pi2))]->State2one(pi1,pi2,pi3,p,a)
[] PR1;[central(pi2)]->State(pi1,pi2,pi3,none,a)
  [] [not(central(pi2))]->State2one(pi1,pi2,pi3,p,a)
[] PR2;State2one(pi1,pi2,pi3,p,a)
[] A1;State2one(pi1,pi2,pi3,p,on)
[] A2;State2one(pi1,pi2,pi3,p,a)
[] AD1;State2one(pi1,pi2,pi3,p,off)
[] AD2;State2one(pi1,pi2,pi3,p,a)
[] (s!pi1!pi2!pi3!p!a;State2one(pi1,pi2,pi3,p,a))
[] State2one(pi1,pi2,pi3,p,a)
[] exit)
endproc
```

Again, the process of making the LOTOS description has forced us to consider just the kinds of details not covered by the requirements. Recall that priority cannot be given to the pilot without control, unless the disengaged stick has been centralised. But, for example, if a pilot has absolute control, and the other pilot depresses his/her priority button *when* his/her stick is centralised, then does the other pilot immediately gain control, without the first pilot releasing his/her priority button? (We conclude that the answer is yes.)

Moreover, what happens when a pilot who has absolute control releases his/her priority button? We conclude that this should have no effect on the overall priority when the other pilot's stick is not centralised. The absolute priority is usually only relinquished after actions from both pilots (assuming that the pilot not in control has to centralise his/her stick). This reflects, we think, the idea that the timing constraint is there to alleviate the need for a pilot to depress the priority button for long periods of time, and so after a period of time, depression or release of that button, in the absence of any centralising movements to the other stick (e.g. the other pilot has had a heart attack!) conveys no information. However, an overidden pilot can gain control from one in absolute control if, after 30 seconds, he/she centralises his/her stick *and* presses his/her priority button.

The `State` process is where the mode is determined. If one pilot has sole control, then this involves determining whether or not that pilot has *absolute* control. This is done by comparing the current time with the last time a priority button was engaged.

```
process State
(pi1,pi2,pi3:pcontrols,p:priority,a:autopilot):=
   [p eq none] -> State1(pi1,pi2,pi3,p,a)
[] [not(p eq none)] ->
     (now?time:Nat; rb?pt:Nat;
       ([(p eq one) and ((time - pt) gt 30)]
       -> State2one
    [] [(p eq one) and ((time - pt) lt 31)]
       -> State1(pi1,pi2,pi3,p,a)
    [] [(p eq two) and ((time - pt) gt 30)]
       -> State2two
    [] [(p eq two) and ((time - pt) lt 31)]
       -> State1(pi1,pi2,pi3,p,a)
endproc
```

Finally, the synchronisation lists in the overall controller process are amended.

```
process Controller
(pi1,pi2,pi3:pcontrols,
 p:priority,a:autopilot,time,pt:Nat):=
   State(pi1,pi2,pi3,p,a)
     |[all events]|
   Pilots
     |[t,s]|
   FCC[s,signals,t]
     |[t,now]|
   Clock[t,now](time)
     |[rb,wb]|
   Button[rb,wb](pt)
endproc
```

6 Analysis

This design was considerably more difficult to develop and analyse than the basic design. The use of an automated testing and simulation tool such as LOLA was crucial as timing constraints are notoriously difficult to get right, and this example was no exception. Moreover, with the additional events, the state space has grown enormously and some degree of automation is essential.

The properties to consider remain as before, with the addition of a variety of properties about the behaviour when a pilot gains and loses absolute control. Again, we do not give full details here, but as an example, we describe how we can construct the test: if pilot 1 is in priority for more than 30 seconds, pilot 2's stick is not centralised, and pilot 2 depresses his/her priority button, then pilot 1 is still in priority. We define a process which expands into a choice between instances of the form:
$t; P1; (t;)^n F1; (t;)^m; P2; s?pi1, pi2, pi3!one!a,$
where $n + m \geq 30$.
We consider only one stick event after the `P1` (and it doesn't matter which one it is) because we have already shown that stick events cannot affect the status of the priority or autopilot. Also, we do not constrain the occurrences of button read or writes in the test; therefore it is important to exclude the events `rb` and `wb` from the synchronisation list between the test and the controller.

7 Discussion

Until recently, LOTOS has been used mainly for descriptions in the OSI (open systems interconnection) context and the telecommunications field (some other applications are reported in [12, 6]). At first, this seemed a new application area for LOTOS, but in hindsight, this problem has many characteristics of a protocol and LOTOS has been ideal for articulating many of the design decisions. It has allowed, indeed forced us to think carefully about the interaction between pilots and in particular, how and when priority is gained and relinquished. This study is a preliminary investigation of the problem, and future work will consider other possible protocols for this system.

8 Conclusions

LOTOS has allowed us to develop a constraint-oriented design description for this problem. This approach *invites* us to develop and test the important details of the components, and then to consider their interactions. The resulting design is one which is close enough to the implementation design, yet abstract enough to allow for rigorous reasoning using automated tools. We are able to conclude, with some confidence, that we have a good, safe design.

Acknowledgements

We thank Tom Melham for his comments.

Addresses for correspondence

M. Thomas, Dept. of Computing Science, B. Ormsby, Dept. of Aerospace Engineering, University of Glasgow, Glasgow G12 8QQ, Scotland.
email: muffy@dcs.gla.ac.uk, bowen@dcs.gla.ac.uk.

References

[1] L. M. Barroca and J. A. McDermid. Formal Methods: Use and Relevance for the Development of Safety-Critical Systems. *The Computer Journal*, 35(6), 1992.

[2] T. Bolognesi and E. Brinksma. Introduction to the ISO Specification Language LOTOS. In P.H.J. van Eijk, C.A. Vissers, and M. Diaz, editors, *The Formal Description Technique LOTOS*, pages 23–76. Elsevier Science Publishers B.V. (North-Holland), 1989.

[3] Hubert Garavel and René-Pierre Hautbois. *Experimenting LOTOS in Aerospace Industry*, chapter 11. Amast Series in Computing. World Scientific, 1994. To appear.

[4] Fly-by-wire controls: The new airliner standard. *International Civil Aviation Authority Bulletin*, pages 19–22, March 1988.

[5] International Organisation for Standardisation. *Information Processing Systems — Open Systems Interconnection — LOTOS — A Formal Description Technique Based on the Temporal Ordering of Observational Behaviour*, 1988.

[6] C. Kirkwood and M. Thomas. Experiences with LOTOS Verification: A Report on Two Case Studies, 1994. Submitted for publication.

[7] B. Littlewood. The Need for Evidence from Disparate Sources to Evaluate Software Safety. In T. Anderson F. Redmill, editor, *Directions in Safety-Critical Systems*, Safety Critical Systems Club, 1993.

[8] J. McDermid. Issues in the development of safety-critical systems. In T. Anderson F. Redmill, editor, *Safety Critical Systems: Current issues, techniques and standards*, Safety Critical Systems Club. Chapman and Hall, 1993.

[9] R. Milner. *A Calculus of Communicating Systems*. LNCS 92. Springer-Verlag, 1980.

[10] J. Rushby. Formal Specification and Verification for Critical Systems: Tools, Achievements, and Prospects. Technical Report EPRI TR-100294, Electric Power Research Institute, January 1992.

[11] J Rushby. Formal Methods and the Certification of Critical Systems. Technical Report CSL-93-7, SRI International, December 1993.

[12] M. Thomas. The Story of the Therac-25 in LOTOS. *High Integrity Systems Journal*, 1(1):3–15, 1994.

[13] K. J. Turner, editor. *Using Formal Description Techniques: An Introduction to ESTELLE, LOTOS and SDL*. Communication and Distributed Systems. Wiley, 1993.