Verifying Fran programs

Simon Thompson
Computing Laboratory, University of Kent

19 April 1999

Abstract

Fran is a system for functional reactive animation. This note explores the way that some of the features of Fran can be interpreted as logical formulas, referring in particular to the example of a lift animation written in Fran.

[First written November 1998; revised February and April 1999.]

1 Introduction

Fran [EH97, PEL97] extends Haskell [HP99], giving a library for functional reactive animations. Animations in Fran evolve in time, so that it seems natural to reason about Fran programs using a temporal logic [GHR94, Eme90]. This note explores how this can be done, examining in particular the example of the lift first introduced in [Tho99].

Indeed, the paper does more than explain how logic can be used to reason about particular Fran programs. It shows how certain Fran programs can be translated into logical formulas, and Fran operators into logical operations.

Obviously other approaches would be possible, and in general it remains to be seen whether this is the form of calculus most suited to reasoning about Fran systems, or whether a calculus based on functions and lists of event occurrences or indeed some other foundation is the most suitable. Certainly a function-based approach would be closer to the denotational semantics of Fran [EH97].

The paper begins by revising some ideas from temporal logic, before showing how elements of Fran can be given a logical interpretation. This is followed by a logical translation of many of the features of the lift example. Section 5 explores in more detail how to axiomatise the properties of the position of the lift, which are given by integrating its velocity. The paper then discusses a number of invariants which the behaviour satisfies, before proving the 'liveness' condition which characterises the active behaviour of the lift.

In modelling Fran logically we have assumed an infinite rate of sampling, so that the `predicate` operation will generate an event occurrence immediately a predicate becomes true; we explore the consequences of this in Section 9.

2 Elements of temporal logic

A temporal logic involves propositions and predicates whose truth values depend not only on logical and term valuations, but also upon a time parameter. A formula is interpreted relative to a semi-infinite closed interval, with the start point providing the 'world' in which values of atomic terms are supplied. Propositional and predicate operators are applied ('pointwise') in the usual way, with the novelty lying in the temporal modalities. The modalities we use we $\Box$, $\Diamond, \mathcal{U}^0$ and $\mathcal{U}$ (for weak and strong 'until').

- The formula $\Box\phi$ will be true in an interval if ϕ is true on all (non-empty) final segments of the interval, that is ϕ holds at all points in the future.
- The formula $\Diamond\phi$ will be true in an interval if ϕ is true on some (non-empty) final segment of the interval, that is ϕ holds at some point in the future.
- The formula $\phi\mathcal{U}^0\psi$ holds on an interval if ϕ holds at all points in the future until ψ holds; this includes the case where ψ never holds, and so $\Box\phi$ holds.
- The formula $\phi\mathcal{U}\psi$ holds if $\phi\mathcal{U}^0\psi$ holds and moreover ψ holds at some point in the future. Formally, $\phi\mathcal{U}\psi$ is equivalent to $\phi\mathcal{U}^0\psi \wedge \Diamond\psi$ and $\phi\mathcal{U}^0\psi$ is equivalent to $\phi\mathcal{U}\psi \vee \Box\phi$.

we revisit the semantics of the 'until' operations later in the paper.

The modalities can also be indexed with positive bounds to express bounded modalities: 'ϕ holds at all points in the next t time units'; 'ϕ holds until ψ holds, and moreover this happens within t time units', and so forth.

This introduction deliberately glosses over many details of temporal logic; more thorough coverage of temporal logics is given in [EH97] and [PEL97].

3 Towards a temporal logic for Fran

This section introduces the logical reading of Fran.

3.1 Behaviours and events

The Fran model addresses both continuous-time behaviour – by means of `Behaviours` – and also discrete events, through its `Event` types. Moreover, the discrete and continuous behaviours will in general be mutually dependent. How can we describe the system we have built here, and systems implemented in Fran in general?

How are Fran-defined values to be described in the logic? We shall take the view that Boolean behaviours (`BoolB`) are atomic temporal formulas[1] and we will also read events (of type `Event ()`) as propositions which are true exactly at their occurrence points.

General behaviours can be compared using equality and other Boolean operations to form atomic propositions.

[1]This neglects the issues of termination; see [Tho95] for a general discussion of the axiomatisation of a functional programming language.

3.2 A logical reading of the constructs of Fran

Under this interpretation, a number of the constructs of Fran are given a logical reading.

- The 'event merge' operation, `.|.`, is seen as logical disjunction over events of type `Event ()`.
- The function `whenE` is interpreted as logical conjunction over `Event ()` and Boolean behaviours.
- Because of these two identifications, it can be seen that `.|.` distributes over `whenE`, and vice versa, at least when applied over `Event ()`.
- Under certain circumstances the `predicate` function is the identity. In general, the event `predicate(c)` is supposed to occur precisely at the start of a period in which `c` holds; if the condition `c` is transitory (see Section 6.1 of [Tho99]) like the conditions `arriveTop` and `arriveBottom` (*op. cit.*), then the event given by applying `predicate` to the condition can be identified with the condition itself.
- A behaviour given by a stepper can be characterised using $\mathcal{U}^0$. Take the example of

  ```
  upPending :: Behavior Bool

  upPending = stepper False (setUp .|. resetUp)

  setUp   = upButton   -=> True
  resetUp = goUp       -=> False
  ```

  This will have the properties that

  $\Box$ (upButton $\Rightarrow$ (upPending $\mathcal{U}^0$ goUp))
  $\Box$ (goUp $\Rightarrow$ (notB upPending $\mathcal{U}^0$ upButton))

  The weak until operator, $\mathcal{U}^0$, has to be defined with some care here. $\phi\,\mathcal{U}^0\psi$ is taken to be true if ϕ is true at all time points *after the start point* of the relevant interval up to and including the first point at which ψ is true. (In the case that ψ is true nowhere in the future, ϕ must hold forever.)

  A more general example of a stepper-defined value is given by the logical interpretation of `dy` in Section 4.

- Fran involves a number of lifted operations, such as `==*`, `>*` and so on. The semantics of the logic lifts the usual connectives like $\wedge$ and $\vee$ over time, and so we shall identify the non-lifted and lifted forms in what follows, using `==` for `==*` and so forth. The meaning will generally be clear from the context in which the operation is used.

4 A logical reading of the lift program

With the interpretations of the previous section it is possible to give a logical interpretation of most aspects of the lift program from [Tho99]. The external events `upButton` and `downButton` necessarily remain uninterpreted; in the next section we interpret the integration given by `atRate`

image, xPos: uninteresting

yPos: the integral of dy; see Section 5

dy: governed by a stepper

$\Box$ (goUp $\Rightarrow$ (dy ==* upRate $\mathcal{U}^0$ (stop $\vee$ goDown))) (dy.1)
$\Box$ (goDown $\Rightarrow$ (dy ==* downRate $\mathcal{U}^0$ (stop $\vee$ goUp))) (dy.2)
$\Box$ (stop $\Rightarrow$ (dy ==* 0 $\mathcal{U}^0$ (goDown $\vee$ goUp))) (dy.3)

atBottom, atTop, stopped, waitingBottom, waitingTop:

```
atBottom = yPos <=* lower
atTop    = yPos >=* upper
stopped  = (dy ==* 0)

waitingBottom = atBottom &&* stopped
waitingTop    = atTop    &&* stopped
```

arriveBottom, arriveTop:

```
arriveBottom = (atBottom &&* dy <* 0)
arriveTop    = (atTop    &&* dy >* 0)
```

upButton, downButton:

```
upButton      -- unanalysed external events
downButton    -- unanalysed external events
```

eitherButton = upButton $\vee$ downButton

pending, upPending, downPending:

pending = upPending $\vee$ downPending

$\Box$ (upButton $\Rightarrow$ (upPending $\mathcal{U}^0$ goUp)) (upPending.1)
$\Box$ (goUp $\Rightarrow$ (notB upPending $\mathcal{U}^0$ upButton))

$\Box$ (downButton $\Rightarrow$ (upPending $\mathcal{U}^0$ goDown))

```
□ (goDown ⇒ (notB downPending U⁰ downButton))

goDown, goUp, stop:

goDown     = (arriveTop ∧ pending)
               ∨
             (eitherButton ∧ waitingTop)

goUp       = (arriveBottom ∧ pending)
               ∨
             (eitherButton ∧ waitingBottom)

stop       = (arriveTop ∨ arriveBottom) ∧ notB pending
```

5 Axiomatising `atRate`

In this section we look at some simple properties of integrals. We assume throughout that `yPos` is defined to be `lower + atRate dy u`, so that `yPos` is the integral of `dy`, or alternatively `dy` is the derivative of `yPos`.

5.1 Integrals and growth

The crucial property here is that if `dy` is bounded away from zero, then its integral, `yPos`, will grow arbitrarily large. Formally,

```
□ (dy >=* k) ⇒ ◇ (yPos >=* v)                          (int.1)
```

and

```
□ (dy <=* -k) ⇒ ◇ (yPos <=* v)                         (int.2)
```

for all `v` and positive `k`. These can be strengthened slightly to give

```
□ (dy >=* k) ⇒ ◇ ((yPos >=* v) ∧ (dy >* 0))             (dy.4)
```

5.2 The lift velocity, `dy`

An immediate consequence of `(dy.4)`, replacing `k` with `upRate` and `v` with `upper` is that

```
□ (dy >=* k) ⇒ ◇ arriveTop                        (arriveTop.2)
```

and propositional reasoning gives

```
arriveTop ⇒ (stop ∨ goDown)                       (arriveTop.3)
```

These two imply that

```
□ (dy >=* k) ⇒ ◇ (stop ∨ goDown)                  (arriveTop.4)
```

and so (dy.1) can be strengthened to

$$\Box \text{ (goUp } \Rightarrow \text{ (dy ==* upRate } \mathcal{U} \text{ (stop } \vee \text{ goDown)))} \qquad \text{(dy.5)}$$

and similarly

$$\Box \text{ (goDown } \Rightarrow \text{ (dy ==* downRate } \mathcal{U} \text{ (stop } \vee \text{ goUp)))} \qquad \text{(dy.6)}$$

where observe that the ‘weak until’s have been replaced by the stronger version.

5.3 Bounding the journey time

We have seen that if dy is bounded away from zero, then its integral, yPos, will grow arbitrarily large. If it is also bounded above, then it is possible to give a lower bound on the time taken to grow to a particular size. In other words, for all positive k and all v we can find t so that

$$\Box \text{ (dy >=* k) } \Rightarrow \Diamond_{\geq \mathtt{t}} \text{ (yPos >=* v)} \qquad \text{(int.3)}$$

and

$$\Box \text{ (dy <=* -k) } \Rightarrow \Diamond_{\geq \mathtt{t}} \text{ (yPos <=* v)} \qquad \text{(int.4)}$$

This has the effect of allowing us to strengthen (dy.5) and (dy.6) to

$$\Box \text{ (goUp } \Rightarrow \text{ (dy ==* upRate } \mathcal{U}_{\geq t} \text{ (stop } \vee \text{ goDown)))} \qquad \text{(dy.7)}$$

and similarly

$$\Box \text{ (goDown } \Rightarrow \text{ (dy ==* downRate } \mathcal{U}_{\geq t} \text{ (stop } \vee \text{ goUp)))} \qquad \text{(dy.8)}$$

5.4 Bounding the integral

If the integrand is bounded and piecewise continuous, then the integral will be continuous. A consequence of this in the running example will be

$$\begin{array}{ll}
\Box \text{ (goUp } \wedge \text{ (lower <=* yPos <=* upper)} & \\
\quad \Rightarrow \text{ ((lower <=* yPos <=* upper) } \mathcal{U}_{\geq t} \text{ (stop } \vee \text{ goDown)))} & \text{(yPos.1)} \\
\Box \text{ (goDown } \wedge \text{ (lower <=* yPos <=* upper)} & \\
\quad \Rightarrow \text{ ((lower <=* yPos <=* upper) } \mathcal{U}_{\geq t} \text{ (stop } \vee \text{ goUp)))} & \text{(yPos.2)} \\
\Box \text{ (stop } \wedge \text{ (lower <=* yPos <=* upper)} & \\
\quad \Rightarrow \text{ ((lower <=* yPos <=* upper) } \mathcal{U}^0 \text{ (goDown } \vee \text{ goUp)))} & \text{(yPos.3)}
\end{array}$$

where t is the bound introduced in the previous sub-section.

6 Time-bounded $\mathcal{U}$ and induction

A putative induction principle uses a hypothesis like

$$\Box \text{ (A } \Rightarrow \text{ P } \mathcal{U} \text{ A)}$$

to give the conclusion

```
□ (A ⇒ □ P)
```

since we can generate an infinite sequence of intervals over which the formula P $\mathcal{U}$ A is true. However, this is not valid since the intervals might form a Zeno sequence, and thus not exhaust an infinite interval.

One way to make this valid is to have a **bounded** version of $\mathcal{U}$ in which there is a lower bound to the interval over which the P holds. If we have a constant k greater than zero so that

$$\Box\ ((A \wedge P) \Rightarrow P\ \mathcal{U}_{\geq k}\ A)$$

then we *can* infer the conclusion

```
□ ((A ∧ P) ⇒ □ P)
```

In a similar way, if we know that (yPos.1), (yPos.2) and (yPos.3) then we can infer by a similar argument that

```
□ ((goUp ∨ goDown ∨ stop) ∧ (lower <=* yPos <=* upper)
      ⇒ □ (lower <=* yPos <=* upper))
```

Note in this case that an infinite sequence must contain infinite numbers of intervals given by goUp and goDown, and so there will be an infinite number of intervals of length at least k, which guarantees the result.

We can also infer that

```
□ ((lower <=* yPos <=* upper) ⇒ □ (lower <=* yPos <=* upper))
```

by noting that prior to any of the actions goUp, goDown and stop, the value of dy is zero and thus yPos is constant. As the starting position of the lift is lower, we finally deduce that

```
□ (lower <=* yPos <=* upper)
```

which states that the lift remains between the upper and lower positions throughout its execution. In the next section we look at other properties like this.

7 Invariants

At the end of the previous section we established that the lift stays between the upper and lower positions throughout its operation.

```
□ (lower <=* yPos <=* upper)                              (I1)
```

This is the first of a number of *invariant* properties which hold for the lift.

The second invariant concerns the value of dy. This is also proved by an induction as above, as indeed is the third invariant, which states that the lift is only stationary when it is in the upper or the lower position.

$\square$ (dy ==* upRate $\vee$ dy ==* downRate $\vee$ dy ==* 0) (I2)
$\square$ (dy ==* 0 $\Rightarrow$ (y ==* upper $\vee$ y ==* lower)) (I3)

Putting the two together, we have an invariant on the behaviour of the lift. At any point it is either moving up, moving down or stationary at the top or the bottom of its range:

$\square$ (dy ==* upRate $\vee$ dy ==* downRate $\vee$ waitingBottom $\vee$ waitingTop) (I4)

8 Verification conditions

This section examines the top-level requirement on the lift's behaviour, and gives a proof of that result.

8.1 The top-level requirement

The top level requirement of the lift can be stated as a conjunction of two conditions

$\square$(upButton $\Rightarrow$ $\diamond$(atBottom $\wedge$ $\diamond$arriveTop)) (P1)
$\square$(downButton $\Rightarrow$ $\diamond$(atTop $\wedge$ $\diamond$arriveBottom)) (P2)

that is 'a request to travel is eventually discharged'. For instance, (P1) states that after a request to travel upwards then eventually the lift is at the bottom (it might already be there) and after that it arrives at the top (atTop could equally well be used here). In the remainder of this section we present a proof of (P1); the proof of (P2) is similar.

8.2 Case analysis

We prove the requirement by making a case split over the cases in (I4).

Case 1: waitingBottom

Suppose that the lift is waiting at the bottom when the button is pressed. Since the property waitingBottom implies atBottom, and this in turn implies $\diamond$atBottom, the first part of the conjunction in (P1) holds.

Now, waitingBottom $\wedge$ upButton implies goUp, and so, by reasoning similar to that proving (yPos.1), we have $\diamond$arriveTop, as required to complete the proof of (P1).

Case 2: waitingTop

By reasoning dual to that in Case 1, we argue that after upButton being pressed then

$\diamond$arriveBottom (2.1)

which has the consequence that $\diamond$atBottom, the first half of (P1).

In order to show that eventually after this the lift will arrive at the top, it is sufficient to derive goUp, as then we can reason as in Case 1. goUp will be a consequence of (2.1) if we can also show that upPending holds on arrival at the lower floor, but this itself is a consequence of (upPending.1) by means of a standard invariant argument.

Case 3: `dy ==* downRate`

The argument in this case is a simple variant of Case 2.

Case 4: `dy ==* upRate`

This case proceeds by combining the argument in Case 3 with its dual, and completes the proof in all cases.

9 Discussion

The analysis here glosses over some issues, particularly to do with taking an infinite sampling rate. We introduce these questions now.

Consider the behaviour of the variable `upPending`, a Boolean which records whether or not an upward journey is pending. The logical reading of this states

$$\Box\ (\texttt{upButton} \Rightarrow (\texttt{upPending}\ \mathcal{U}^0\ \texttt{goUp})) \qquad (\texttt{upPending.1})$$
$$\Box\ (\texttt{goUp} \Rightarrow (\texttt{notB upPending}\ \mathcal{U}^0\ \texttt{upButton})) \qquad (\texttt{upPending.2})$$

Informally, `upPending` holds up to the point when the event `goUp` occurs, when `upPending` becomes false. In order to avoid a logical inconsistency, it appears that `upPending` should only be false *after* the occurrence of `goUp`, rather than becoming true immediately at that point. This will affect the semantics of the $\mathcal{U}^0$ construct used in modelling the `stepper` here.

A second question concerns the general behaviour of the `predicate` function. Consider the event

```
predicate( yPos >* lower )
```

The predicate '`yPos >* lower`' will hold over a semi-open interval of time, and in particular the interval itself does not contain the infimum for the times at which the event might occur. This provides a challenge for the general logical modelling of Fran systems, but does not present a problem for the lift model as given here.

References

[EH97] Conal Elliott and Paul Hudak. Functional Reactive Animation. In *Proceedings of the 1997 ACM SIGPLAN International Conference on Functional Programming (ICFP97)*. ACM Press, 1997.

[Eme90] E Allen Emerson. Temporal and Modal Logic. In Jan van Leeuwen, editor, *Handbook of Theoretical Computer Science, Volume B: Formal Models and Semantics*. Elsevier/MIT Press, 1990.

[GHR94] Dov M. Gabbay, Ian Hodkinson, and Mark Reynolds. *Temporal Logic: Mathematical Foundations and Computational Aspects*, volume 1. Oxford University Press, 1994.

[HP99] John Hughes and Simon Peyton Jones, editors. *Report on the Programming Language Haskell 98*. `http://www.haskell.org/report/`, 1999.

[PEL97] John Peterson, Conal Elliott, and Gary Shu Ling. Fran Users' Manual. Available from `http://www.haskell.org/fran`, 1997.

[Tho95] Simon Thompson. A Logic for Miranda, Revisited. *Formal Aspects of Computing*, 7, 1995.

[Tho99] Simon Thompson. A functional reactive animation of a lift using Fran. Submitted for publication in J. Functional Programming; available from `http://www.cs.ukc.ac.uk/people/staff/sjt/Fran`, 1999.

Appendix: The complete program for the two floor lift

```
liftSim :: User -> ImageB

liftSim u

  = moveXY xPos yPos image

    where

    image :: ImageB

    image = stretch 0.3 circle

    xPos,yPos,dy :: RealB

    xPos = constantB 0
    yPos = lower + atRate dy u
    dy   = stepper 0 setRate

    setRate :: Event Double

    setRate = stop      -=> 0
              .|.
              goUp      -=> upRate
              .|.
              goDown    -=> downRate

    atBottom, atTop, stopped, waitingBottom, waitingTop :: BoolB

    atBottom = yPos <=* lower
    atTop    = yPos >=* upper
    stopped  = (dy ==* 0)

    waitingBottom = atBottom &&* stopped
    waitingTop    = atTop    &&* stopped

    arriveBottom, arriveTop :: Event ()

    arriveBottom = predicate (atBottom &&* dy <* 0) u
    arriveTop    = predicate (atTop    &&* dy >* 0) u
```

```
upButton, downButton, eitherButton :: Event ()

upButton     = lbp u
downButton   = rbp u
eitherButton = upButton .|. downButton

upPending, downPending, pending :: Behavior Bool

pending     = upPending ||* downPending

upPending   = stepper False (setUp   .|. resetUp)
downPending = stepper False (setDown .|. resetDown)

setUp, setDown :: Event Bool

setUp   = upButton   -=> True
setDown = downButton -=> True

resetUp, resetDown :: Event Bool

resetUp   = goUp   -=> False
resetDown = goDown -=> False

goDown, goUp, stop :: Event ()

goDown      = arriveTop    `whenE` pending
              .|.
              eitherButton `whenE` waitingTop

goUp        = arriveBottom `whenE` pending
              .|.
              eitherButton `whenE` waitingBottom

stop        = (arriveTop .|. arriveBottom) `whenE` notB pending
```