An Approach to Modelling and Refining Timing Properties in B

Michael Butler
University of Southampton
Highfield, Southampton SO17 IBJ, United Kingdom

Jérôme Falampin
Siemens Transportation Systems
48–56, rue Barbs, BP 531, 92542 Montrouge Cedex, France

7 December 2001

1 Introduction

The work described in this extended abstract is being undertaken as part of the EU-funded MATISSE project (IST-1999-11435). One of the major case studies of MATISSE involves the application of the B Method [2] to a railway control system. The emphasis of the work is on system-level modelling and analysis. This means we are not just modelling pieces of control software in B, but we are using B to model relevant aspects of an entire network. For example, a system-level model would include physical connections between track sections, the positions of trains in terms of the sections they currently occupy, and under what system-level conditions the emergency brakes should be applied to ensure safety. This system-level model has been decomposed and refined into distributed trackside and on-board controllers with messages passing between them. This allows us to derive the software specifications for the individual controllers in a way that increases our confidence that the combination of the software and physical components achieve the desired system-level safety properties.

While taking a systems approach to the development of the railway controllers, it became clear that it was important to model timing constraints in some form. For example, without any model of time we could represent a requirement on the emergency brakes in B in one of two forms:

- *The brakes can be applied when a train is in a restrictive section.*
- *Immediately when the section a train is in becomes restrictive, the brakes must be applied.*

The first of these is too weak as the brakes might not be applied in time to avoid a collision. The second is too strong as there is always a short delay between a section going restrictive and the on-board system being informed of this by the trackside system. We wish to be able to express a requirement of the form:

- *Within X milliseconds of a section becoming restrictive, the emergency brakes of a train on that section are applied.*

There are also other reasons why timing is important. In refinements of the system-level model, the messages sent between the trackside and the on-board systems are encrypted and time stamped to counteract communications errors and malicious tampering, and there are timeout mechanisms in case of message delays.

A practical constraint was that we wanted to prove the correctness of our refinements using Atelier-B. This meant we needed to avoid going down the route of trying to extend B in any way as

this would have required extensions to the tool and possibly extensions to the semantics. Instead, we needed to use B as supported by Atelier-B.

Some familiarity with B is assumed in the rest of this extended abstract.

2 Approach

The approach we take to modelling timing constraints is to include a clock variable representing the current time and a operation which advances this variable. We ensure the timing constraints are satisfied by preventing the clock variable from progressing to a point at which the required properties would be violated. It would seem that we could always satisfy the properties by preventing time from progressing. However, we assume that in the real system time cannot be prevented from progressing and thus it is an obligation on the final implementation to ensure that the timing properties are always satisfied in time. This is a fairly standard approach to dealing with time in formalisms. For example, this is similar to the approach taken by Abadi and Lamport [1]. Unlike Abadi and Lamport, we are using a discrete model of time rather than a continuous one. A discrete model is sufficient for our purposes since we are interested in ensuring that certain properties hold within fixed time bounds and since the control of the real system is based on fixed time cycles. When we say that something happens at a certain time t, what we mean is that it happened within time period t.

Consider the timing property on the emergency brakes again: *within X milliseconds of a section becoming restrictive, the emergency brakes of a train on that section are applied.* To represent this constraint in B, we include a variable which records the most recent time at which a section went restrictive, *rtime*. Then the requirement may be formalised in the following form (t is the current time):

$$Prop(t) \;=\; t \geq rtime + X \;\Rightarrow\; braking = true$$

The operation which progresses time is then guarded by the condition that the property holds in the new time period:

$$NextTime \;=\; \text{SELECT}\;\; Prop(t+1)\;\text{THEN}\;\; t := t+1\;\text{END}$$

Ultimately the real system will have to ensure that $Prop(t+1)$ does indeed hold by the time $t+1$ is reached. The first refinement that we present here goes some way towards ensuring that the timing property holds in time. The refinement introduces message passing between the trackside and the on-board systems along with a timeout mechanism so that, if messages are not received within a certain time period, the emergency brakes are applied. The second refinement introduces time stamps in the messages, along with an operation modelling explicit attempts at message tampering by an intruder.

3 Specification

We present a simplified version of the system-level specification. The state variables are defined as follows (CDV represents the set of track sections):

```
res : POW(CDV) &              /* set of restrictive sections */
pos : TRAIN --> CDV &         /* each train has a single position */
cur : NATURAL &               /* current time interval */
rtime : CDV --> (0..cur) &    /* time when section goes restrictive */
atime : TRAIN --> (0..cur) &  /* time when train entered current position */
braking <: TRAIN &            /* set of braking trains */
```

The property $BrakingProperty(cur)$ should be invariant, where $BrakingProperty(i)$ is defined as follows:

```
BrakingProperty(i) ==
 !tt.( tt:TRAIN  &  pos(tt):res  &
       atime(tt) < rtime(pos(tt))  &  rtime(pos(tt)) + XX <= i
       =>  tt:braking  )
 &
 !tt.( tt:TRAIN  &  pos(tt):res &
       atime(tt) >= rtime(pos(tt))  &  atime(tt) + XX <= i
       =>  tt:braking  )
```

The braking property has two cases. If the train has arrived in the section before the section went restrictive, then the time bound is relative to the time at which the section went restrictive. If the train has arrived in the section after the section went restrictive, then the time bound is relative to the time at which the train arrived.

Train arrival and sections going restrictive are modelled by the following operations:

```
Arrive(t1,c1) =
  PRE  t1:TRAIN  &  c1:CDV  THEN  pos(t1) := c1  ||  atime(t1) := cur  END
Restrict(c1) =
  PRE  c1:CDV  THEN  res := res \/ {c1}  END
```

Note that we have simplified things considerably here. In our larger model, sections are restrictive when certain other sections are occupied so that the *Arrival* operation causes relevant other sections to go restrictive. Also in the larger model, a train cannot arrive at an arbitrary section.

The operation which increases time is guarded by the braking property holding on the next time interval:

```
NextTime=
  SELECT  BrakingProperty(cur+1)  THEN  cur:=cur+1  END
```

In case the braking property would not hold in the next time interval, it can always be made to hold by some invocations of the operation which models application of the brakes:

```
Brake(t1) =
  PRE  t1:TRAIN  THEN  braking:=braking \/ {t1}  END
```

This operation is always enabled. In the next section we strengthen the guard (which is allowed in B refinement) so that it is enabled when a train has received a restrictive message or when a timeout has occurred.

4 Messages and Timeouts

In this section we describe a first refinement that uses message passing to transfer information about restrictivity. We introduce variables to model this, including variables modelling the time at which the most recent message to a train was sent and received:

```
  msg : TRAIN +-> BOOL &        /* messages in transit to trains */
  flag : TRAIN --> BOOL &       /* value of most recently received message */
  mstime : TRAIN --> (0..cur) & /* message send time */
  mrtime : TRAIN --> (0..cur)   /* message receive time */
```

Note that msg is a partial function. When $t \in dom(msg)$, then $msg(t)$ has been sent to train t but not yet received by t. The value $TRUE$ for $msg(t)$ indicates that the train should stop.

We assume that the trackside system continually sends messages to a train every MIT time units. In this way a train knows by what time it should expect to receive a message. If it hasn't received a message within the expected time, it applies the emergency brakes. We also assume that there can be a delay in receiving messages and that the maximum expected delay is MDT

time units. Thus a train should wait no longer than $MIT + MDT$ time units in between message receipts before applying the brakes.

The braking property used to guard the *NextTime* operation in the refined system has three parts:

```
BrakingProperty2(i) ==
 !tt.( tt:TRAIN  &  mrtime(tt)+MDT+MIT < i   =>  tt:braking  )
 &
 !tt.( tt:TRAIN  &  tt/:braking  &  tt:dom(msg)   =>  mstime(tt)+MDT >= i   )
 &
 !tt.( tt:TRAIN  &  flag(tt)=TRUE   =>  tt:braking )
```

The first clause says that if it is more than $MDT + MIT$ time units since a train last received a message, then the train should be braking. The second clause says that a message should not be in transit for longer than MDT time units (unless the train is already braking); this ensures that timeliness of a message which is important in the case that the message is permitting the train to continue without braking, i.e., *FALSE* message. The third clause says that when a *TRUE* message has been received, then the train should brake immediately.

Note that these timing constraints are more localised (and thus easier to implement) than those of the system-level specification. The first and third clause represent constraints on the on-board system, while the second represents a constraint on the message transmission mechanism.

We introduce operations modelling the sending of messages by the trackside system and the receipt of messages by the on-board system as follows:

```
SendMsg(t1) =
  SELECT  t1/:dom(msg) & cur<=mstime(t1)+MIT  THEN
    msg(t1) := bool( pos(t1):res )  ||  mstime(t1) := cur
  END

RecvMsg(t1) =
  SELECT  t1:dom(msg)  THEN
    flag(t1) := msg(t1)  ||  mrtime(t1):=cur ||  msg := {t1} <<| msg
  END
```

The *Brake* operation is refined so that it is enabled either when a *TRUE* value has been received or $MDT + MIT$ time units have elapsed since the last message receipt:

```
Brake(t1) =
  SELECT  flag(t1) = TRUE  THEN  braking := braking \/ {t1}
  WHEN  cur >= mrtime(t1) + MDT + MIT  THEN  braking := braking \/ {t1}
  END
```

5 Time Stamping of Messages

In this section we allow for the possibility that an intruder might attempt to tamper with messages. Because messages are encrypted using secret keys, the only form of tampering possible is replay of previously sent messages. We have included an operation explicitly modelling message replay and have shown that the refinement is valid regardless, i.e., the mechanism to counteract message replay attacks is safe.

The *msg* variable is replaced by a variable *tmsg* in which messages are time stamped. We also introduce two further variables, *allmsg* recording all messages sent to date to a train, and *timestamp*, recording the time stamp value of the most recently received message by a train:

```
 tmsg: TRAIN +-> (BOOL*(0..cur)) &
 allmsg : TRAIN --> POW(BOOL*(0..cur)) &
 timestamp : TRAIN --> (0..cur)
```

The *SendMsg* operation is refined so that a pair consisting of a boolean and a time stamp is sent to a train, and the message is recorded in *allmsg*:

```
SendMsg(t1) =
  SELECT  t1/:dom(tmsg)  &  cur<=mstime(t1)+MIT
  THEN
    tmsg(t1) := bool( pos(t1):res ) |-> cur  ||
    mstime(t1) := cur ||
    allmsg(t1) := allmsg(t1) \/ {  bool( pos(t1):res ) |-> cur  }
  END
```

Although not explicitly modelled, we assume that each message is encrypted for a train using a secret key so that it is impossible for an intruder to fake or alter a message. Thus the only form of tampering possible of is for an intruder to replay previously sent messages. This is modelled by the following operation:

```
FakeMsg(t1) =
  ANY m1 WHERE  m1 : allmsg(t1)  THEN  tmsg(t1) := m1  END
```

Now, the *RecvMsg* operation only accepts a message if its time stamp is greater than the time stamp of the previously sent message. This means it will not accept replayed messages that are stale and this possibly invalid:

```
RecvMsg(t1) =
  ANY bb,ee WHERE
    bb:BOOL  &  ee:NATURAL &
    t1:dom(tmsg) & (bb|->ee)=tmsg(t1)  &  ee > timestamp(t1)
  THEN
    flag(t1) := bb  ||  mrtime(t1) := cur  ||
    timestamp(t1) := ee  ||  tmsg := {t1} <<| tmsg
  END
```

6 Gluing Invariant and Proof

All the proof obligations associated with the specification and the refinements have been proved using Atelier-B. An essential ingredient when proving a refinement is the gluing invariant describing the relationship between the variables of the abstract and concrete systems. For the refinements presented here, we took an iterative approach to constructing the gluing invariants. This means that as we attempt to discharge proof obligations interactively using Atelier-B, we discover new assumptions that are required. We check informally that these assumptions are a reasonable reflection of the requirements. If so, the invariant is strengthened using these assumptions and the proof is attempted again. Typically, these new invariant clauses give rise to further proof obligations which may in turn require further new assumptions. Several iterations of invariant strengthening may be required before an invariant strong enough to discharge all the proof obligations is arrived at. For both the refinements presented here, three iterations of invariant strengthening were required. The final invariant for the first refinement results in 55 proof obligations, 34 of which were proved automatically and 21 interactively. The same figures for the second refinement were 43, 21 and 22 respectively.

References

[1] M. Abadi and L. Lamport. An old-fashioned recipe for real time. *ACM Transactions on Programming Languages and Systems*, 16(5):1543–1571, 1994.

[2] J.R. Abrial. *The B-Book: Assigning Programs to Meanings*. Cambridge University Press, 1996.