Treatment of Time Constraints in Z – A Case Study

Maritta Heisel
Technische Universität Berlin
FB Informatik – Softwaretechnik
Franklinstr. 28-29, Sekr. FR 5-6
D-10587 Berlin, Germany
heisel@cs.tu-berlin.de

Friederike Nickl
Ludwig-Maximilians-Universität München
Institut für Informatik
Leopoldstr. 11B
D-80802 München, Germany
nickl@informatik.uni-muenchen.de

November 13, 1995

1 Introduction

This is the first version of a specification of the transit node in Z. It is more pragmatic than the informal specification given in [MPB+89]. It basically specifies the transit node as one would probably implement it. Not each message has its own private timer, but there is a global system time. Incoming messages are stamped with their arrival time, and if the current time is greater than the arrival time plus a predefined constant T, the message will not be sent any more to its given address but will be marked as faulty and be directed to the control port out.

From time to time, an operation *MarkFaults* is invoked that checks for each message if it is already too old, and if so, marks it as faulty. This means that the messages cannot get faulty spontaneously. If we assume a sequential implementation of the transit node, this in turn implies that it is possible that there are messages in the node that have exceeded the time limit but are not yet marked as faulty. But even in a parallel implementation this problem would occur because the operation that marks a message as faulty and the operation that regularly sends a message out of the node access the same data. Hence, they cannot really operate in parallel. Only if we would consider very large time grains, so that both operations can be finished within one time unit could we guarantee that a message is marked as faulty as soon as it exceeds the time limit.

To guarantee that each message that is too old will be marked as faulty within some time limit, we introduce a tolerance. Each message for which the current time is larger than the arrival time plus the constant *timoeout* plus the tolerance must be marked as faulty. The tolerance may depend on the number of messages present in the node since the complexity of *MarkFaults* is at least linear in this number. Moreover, it is guaranteed that no message that has exceeded the time limit will be sent. This is checked by the sending operation.

The present specification states that there is a global system time, and that each operation needs a positive amount of time. The requirement that the variable *ct* refers to the internal clock of the target computer is stated informally because this cannot be expressed in Z. For each operation, a function is defined that yields the time its needs for completion, depending on the complexity of the node and the number of messages to be processed.

The papers explores the usability of the specification language Z to model systems that underly real-time constraints. It seems that for the present case study the modeling is adequate, or at least as adequate as it would be possible in other specification languages. In any case, the specification presented here is more realistic than the one given in [MPB+89].

2 Basic Definitions

It has turned out that there is no necessity to explicitly model the control ports. The control port in receives messages that change the infrastructure of the transit node. These messages are

treated by changing the *Infrastructure* component of the transit node. The control port out takes care of faulty messages when they leave the node. "Leaving the node", however, just means that the message is no longer present in the node, and this is modeled accordingly. Hence, the following basic definitions suffice:

$$[DATA, ROUTE, PORT, OUTPORT, INPORT]$$

$$TIME == \mathbb{N}_1$$

$$T : TIME$$
$$tolerance : \mathbb{N} \longrightarrow TIME$$

$$\forall n, m : \mathbb{N} \bullet n \geq m \Rightarrow tolerance\, n \geq tolerance\, m$$

Ports are always bi-directional. The means, each port is associated with an input port and an output port.

$$inport : PORT \rightarrowtail INPORT$$
$$outport : PORT \rightarrowtail OUTPORT$$

Since the system time (called current time, ct) changes it must be modeled as an operation.

CurrentTime
$$ct : TIME$$

Messages that reach the node just consist of a data part and a route that tells the node where to send them.

ExternalMessage
$$d : DATA$$
$$r : ROUTE$$

When messages are stored in the node for further treatment, they receive a time stamp.

InternalMessage
$$d : DATA$$
$$r : ROUTE$$
$$arr\_time : TIME$$

3 The Global System State

The transit node consists of two parts, the infrastructure and the messages waiting to be passed on. The infrastructure consists of a number of data ports in and the same number of data ports out. Moreover, there is a relation *routemap* that shows which data port out can be used for which route.

Infrastructure
$$ports : \mathbb{F}\, PORT$$
$$inports : \mathbb{F}\, INPORT$$
$$outports : \mathbb{F}\, OUTPORT$$
$$routemap : ROUTE \leftrightarrow OUTPORT$$

$$inports = \{p : ports \bullet inport\, p\}$$
$$outports = \{p : ports \bullet outport\, p\}$$
$$\mathrm{ran}\, routemap \subseteq outports$$

The messages to be processed are represented by the schema *MessageDistribution*. The waiting non-faulty messages are associated with a suitable outport, represented by the function *dest* (for destination). There is also a set of faulty messages. A message is either faulty because the time limit is exceeded or because there is no outport corresponding to the message's route. The second case causes a problem. Usually, one would state that each faulty message is either timed out or its corresponding route does not belong to dom *routemap*. However, this is not possible because the message's route can be added to the transit node after the message has arrived and been inserted into the faulty message set and before the faulty message has been sent out of the node.

MessageDistribution

$$dest : InternalMessage \pfun OUTPORT$$
$$faulty\_msgs : \mathbb{F}\ InternalMessage$$

The integrity constraints holding for the transit node concern the connection between the infrastructure and the message distribution as well as the time constraints. They can only be stated in the schema representing the entire transit node.

Before we can state these, however, we must make an auxiliary definition because we must define a function on transit nodes that is in turn used in the integrity contraint.

TR

$$CurrentTime$$
$$Infrastructure$$
$$MessageDistribution$$

The next two functios model the internal complexity of the node, depending mostly on its infrastructure, and the load under which the node works, depending mostly on the number of messages in the node.

$$complexity : TR \longrightarrow \mathbb{N}$$
$$load : TR \longrightarrow \mathbb{N}$$

TransitNode

$$TR$$

$$\forall m : \mathrm{dom}\, dest \bullet$$
$$\quad m.arr\_time + T + tolerance(complexity(\theta TR) + load(\theta TR)) \geq ct \land$$
$$\quad m.r \mapsto dest\ m \in routemap$$

You may wonder why the tolerance does not only get $\#dest$ as its argument. The reason will become clear in Section 6 where we will define a control schema that guarantees the state invariant to be maintained.

Again, we would like to state

$$\forall m : faulty\_msgs \bullet$$
$$\quad (m.arr\_time$$
$$\qquad + T + tolerance(complexity(\theta TR) + load(\theta TR)) < ct \lor$$
$$\quad m.route \notin \mathrm{dom}\, routemap)$$

but because of the reason stated above this is not possible. The initial state is as follows:

InitTransitNode

$$TransitNode'$$

$$ports' = \varnothing$$
$$dest' = \varnothing$$
$$faulty\_msgs' = \varnothing$$

It follows that $inports'$, $outports'$ and $routemap'$ must also be empty. Since dom $dest'$ is empty, too, the initial state fulfills the invariant.

Our aim is to define a control schema (see Section 6 that makes sure that the invariant of the transit node is not violated. If this is the case or not depends on the execution times of the various operations of the transit node. This execution time, in turn, depends on the complexity and load of the node, respectively. Since we cannot define a type *TransitNodeOperation* whose members are exaclty the operations on *TransitNode*, we have to define a seperate function yielding the execution time for each operation we will define in the following sections.

$$
\begin{array}{l}
d\_add\_port, d\_add\_route : \mathbb{N} \longrightarrow TIME \\
d\_mark\_faults, d\_rec\_mess, d\_msg\_send, d\_faulty\_send : \mathbb{N} \longrightarrow TIME \\
\hline
\textbf{let } fct\_set == \{d\_mark\_faults, d\_add\_port, d\_add\_route, d\_rec\_mess, \\
\qquad d\_msg\_send, d\_faulty\_send\} \bullet \\
\quad (\forall f : fct\_set; n, m : \mathbb{N} \bullet f\ n + d\_mark\_faults\ n \leq tolerance\ n \\
\quad \land\ n \geq m \Rightarrow f\ n \geq f\ m)
\end{array}
$$

The constraint given here makes sure that it suffices to execute the operation *MarkFaults* every second time an operation is executed. Otherwise, we always had to execute *MarkFaults* in order to maintain the state invariant, and there would be no time for the operations implementing the intended purpose of the node.

4 Operations Changing the Infrastructure

When a new pair of ports is added, no routes are associated with the outport. This is done in a separate operation. These operations are not particularly interesting. They also will not be used very often in practice.

AddDataPort

$$
\begin{array}{l}
\Delta TransitNode \\
\Xi MessageDistribution \\
p? : PORT \\
\hline
p? \notin ports \\
ports' = ports \cup \{p?\} \\
inports' = inports \cup \{inport\ p?\} \\
outports' = outports \cup \{outport\ p?\} \\
routemap' = routemap \\
ct' = ct + d\_add\_port(complexity(\theta\, TransitNode))
\end{array}
$$

AddRoute

$$
\begin{array}{l}
\Delta TransitNode \\
\Xi MessageDistribution \\
r? : ROUTE \\
ops? : \mathbb{F}_1\ OUTPORT \\
\hline
ops? \subseteq outports \\
routemap' = routemap \cup \{op : ops? \bullet r? \mapsto op\} \\
ports' = ports \\
inports' = inports \\
outports' = outports \\
ct' = ct + d\_add\_route(complexity(\theta\, TransitNode))
\end{array}
$$

5 Operations Changing the Message Distribution

The next operation checks each message in the node and marks it as faulty if necessary.

MarkFaults

$$\Delta TransitNode$$
$$\Xi Infrastructure$$

$$\textbf{let } new\_faulty == \{m : \mathrm{dom}\, dest \mid m.arr\_time + T \leq ct\} \bullet$$
$$(dest' = new\_faulty \lhd dest \land$$
$$faulty\_msgs' = faulty\_msgs \cup new\_faulty)$$
$$ct' = ct + d\_mark\_faults(load(\theta TransitNode))$$

The problem here is that *doing nothing* destroys the global system invariant because time increases no matter if an operation is executed or not. "Doing nothing" is defined by the following schema:

Idle

$$\Delta TransitNode$$
$$\Xi Infrastructure$$
$$\Xi MessageDistribution$$

$$ct' > ct + 1$$

In order to maintain the system invariant, the operation *MarkFaults* must be carried out at least once in each time interval of length *tolerance*, where this tolerance increases with the number of messages in the node. This issue is treated in Section 6.

ReceiveMessage

$$\Delta TransitNode$$
$$\Xi Infrastructure$$
$$ip? : INPORT$$
$$im? : ExternalMessage$$

$$ip? \in inports$$
$$\exists\, im : InternalMessage \mid im.d = im?.d \land im.r = im?.r \land im.arr\_time = ct \bullet$$
$$((\exists\, op : outports \mid im.r \mapsto op \in routemap \bullet$$
$$(dest' = dest \cup \{im \mapsto op\} \land faulty\_msgs' = faulty\_msgs))$$
$$\lor$$
$$(\forall\, op : outports \bullet$$
$$(im.r \mapsto op \notin routemap \land dest' = dest \land faulty\_msgs' = faulty\_msgs \cup \{im\})))$$
$$ct' = ct + d\_rec\_mess(load(\theta TransitNode))$$

Theoretically, it is possible that two messages that are exactly the same arrive at different in ports at the same time. If these messages were associated with different out ports, then the property of *dest* to be a function could be destroyed. This situation, however, seems to be so improbable that we prefer not to take it into account explicitly.

DataMessageSending

$\Delta\, TransitNode$
$\Xi\, Infrastructure$
$op? : OUTPORT$
$om! : ExternalMessage$

$\exists\, im : InternalMessage \mid dest\, im = op? \land im.arr\_time + T < ct \bullet$
$\quad (dest' = dest \setminus \{im \mapsto op?\} \land$
$\quad faulty\_msgs' = faulty\_msgs \land$
$\quad om!.d = im.d \land om!.r = im.r)$
$ct' = ct + d\_msg\_send(load(\theta\, TransitNode))$

This operation takes care not to send a message that is already timed out. Messages leaving the node are modeled as an output of the respective operation. This output can serve as an input for those procedures that actually implement the sending process.

FaultyMessageSending

$\Delta\, TransitNode$
$\Xi\, Infrastructure$
$fm! : ExternalMessage$

$\exists\, ifm : faulty\_msgs \bullet$
$\quad (ifm.d = fm!.d \land ifm.r = fm!.r \land$
$\quad faulty\_msgs' = faulty\_msgs \setminus \{ifm\})$
$dest' = dest$
$ct' = ct + d\_faulty\_send(load(\theta\, TransitNode))$

The next version of the specification will associate a timer with each message and will probably be written in Object-Z. Perhaps it is possible to find a history invariant that allows us to state some nice liveness and fairness properties for the node. This specification given here is in my opinion somewhat more appropriate concerning the modeling of time as the PLUSS specification given in [MPB+89].

6 Specifying Control

We now want to explore how far we can get in Z in specifying the timing constraints the transit node must fulfill. Obviously, we take a pragmatic approach: when the load on the node is heavy, i.e. there are many messages to be handeled, the node may be a bit slower in handling the faulty messages.

The functions *d_add_port* and *d_add_route* depend on the complexity of the node, whereas the other functions depend on its load. These numbers will usually be independent of each other. Hence we must make sure that the tolerance will always be computed with the largest possible number, i.e. the sum of all these numbers, as this is done above.

$Flag ::= yes \mid no$

ControlState

$\Delta\, TransitNode$
$last\_call\_of\_MarkFaults : TIME$

$(last\_call\_of\_MarkFaults = 1 \lor$
$\quad last\_call\_of\_MarkFaults + d\_mark\_faults(load(\theta\, TransitNode)) \leq ct)$
$\forall\, im : \mathrm{dom}\, dest \bullet im.arr\_time + T \geq last\_call\_of\_MarkFaults$

InitControlState

$ControlState'$

$InitTransitNode$

$last\_call\_of\_MarkFaults' = 1$

The initial state fulfills the invariant. A schema name in the predicate part of another schema means that the schema predicate of the imported schema is required to hold.

`muss die Definition von` *flag* `noch von` T `abhaengen?`

Control

$\Delta ControlState$

---

let $margin ==$
$max\{d\_add\_port(complexity(\theta\, TransitNode)),$
$d\_add\_route(complexity(\theta\, TransitNode)), d\_rec\_mess(load(\theta\, TransitNode)),$
$d\_faulty\_send(load(\theta\, TransitNode))\} \bullet$
(**let** $flag ==$
if $ct + margin + d\_mark\_faults(load(\theta\, TransitNode))$
$\geq last\_call\_of\_MarkFaults + tolerance(complexity(\theta\, TransitNode) + load(\theta\, TransitNode))$
then yes
else $no \bullet$
$(flag = yes \Rightarrow MarkFaults \land last\_call\_of\_MarkFaults' = ct)$
$\land$
$(flag = no \Rightarrow$
$((MarkFaults \land last\_call\_of\_MarkFaults' = ct)$
$\lor (((\exists\, p? : PORT \bullet AddDataPort)$
$\lor (\exists\, r? : ROUTE; ops? : \mathbb{F}_1\, OUTPORT \bullet AddRoute)$
$\lor (\exists\, ip? : INPORT; im? : ExternalMessage \bullet ReceiveMessage)$
$\lor (\exists\, op? : OUTPORT; om! : ExternalMessage \bullet DataMessageSending)$
$\lor (\exists\, fm! : ExternalMessage \bullet FaultyMessageSending)$
$\lor Idle)$
$\land last\_call\_of\_MarkFaults' = last\_call\_of\_MarkFaults))))$

To define the control operation, we had to existentially quantify over the input and output variables because the different schemas have different interfaces.

It is now necessary to prove that this control operation guarantees that the state invariant is never violated.

References

[MPB+89] M. Mauboussin, H. Perdrix, M. Bidoit, M.-C. Gaudel, and J. Hagelstein. From an ERAE requirements specification to a PLUSS algebraic specification: A case study. In *Proceedings Meteor workshop, Algebraic Methods II*, number 490 in LNCS. Springer-Verlag, 1989.