JET PROPULSION LABORATORY INTEROFFICE MEMORANDUM
SFB: 363-87-006

AUGUST 17, 1987

ZIP FORWARD

Steve Bellenot

This outlines the main changes between my personal copy of Time Warp and the current version 1.05. There are three main sections: OLD_LVT -- the base line for the past/future division; RDY_2_ROLL -- a way of disconnecting enqueuing messages from the side effects of rollback; and ZIP_FORWARD -- my version of "go_forward". Some other odds and ends are also noted at the end.

OLD_LVT:

For reasons which will become clear later, I added a field to each Ocb struct called old_lvt and use it as a reference to maintain four additional fields in the Ocb struct (num_inq_msg_past, num_inq_msg_future, num_outq_msg_past and num_outq_msg_future). If the message M arrives with receive time < old_lvt, then num_inq_msg_past is decremented(incremented) if M annihilates (enqueues) and similarly with the "future" if receive time >= old_lvt. The "outq" variables keep track of output messages by send time. These num_in(out)q_msg_past(future) variables are updated on the fly and don't seem to have a noticeable effect on the speed of Time Warp. (Global analogs of these variables are also maintained on each node.)

The thought was that these variables would allow one to determine if an object was 1. a slow consumer (num_inq_msg_future large), 2. had gotten way ahead and then was rolled back (num_outq_msg_future large), or 3. ahead of the simulation (num_in(out)q_msg_past large). Large is a relative term which can change as the past is garbage collected.

Clearly we want old_lvt to be as near to LVT as possible. However, there is the time between the arrival of message earlier than LVT and the moment the object is rolled back to that receive time, that lvt isn't the correct base line. It turns out that if states are always saved, then old_lvt = lvt whenever the "runstatus" is READY.

And old_lvt = lvt is always true after a call to roll_it_back. (However, svt < lvt < old_lvt and "runstatus" = READY is possible.)

The motivation for changing go_forward into zip_forward was to make these num_in(out)q_msg_past(future) correct. It was thought to be easier to re-code, then to make the current code count correctly. If these num_in(out)q_msg_past(future) turn out to be uninteresting, then there may be no reason to change go_forward into zip_forward.

RDY_2_ROLL:

The token RDY_2_ROLL is a #defined constant to be used in the Ocb struct field "runstatus". An object is put into the RDY_2_ROLL status by the routine "check_4_rollback" which is called by the enqueuing routines. An object exits the RDY_2_ROLL status when dispatch calls the routine "roll_it_back" to make the object READY again. Thus an enqueuing routine falls off its bottom brace before the side effects of rollback happen.

The routine check_4_rollback is nearly the same as the first few lines of the old rollback routine. If the message's time is less than or equal to lvt, then lvt is set to the message's time and the object is made RDY_2_ROLL. Otherwise, the message's time is greater than lvt, rollback isn't needed and check_4_rollback just returns.

Just before the bottom brace of each entry into Time Warp is a call to dispatch. If next object for dispatch to try to run is RDY_2_ROLL then one could call the old routine rollback (object, object->lvt). However, we call the routine roll_it_back which does zip_forward instead of go_forward. Some care is needed in dispatch since rollback and roll_it_back can change the priority of an object.

ZIP_FORWARD:

The routine go_forward finds the end points "from" and "to" in virtual time and then cancels unmarked output messages in this interval. This can be seen as viewing virtual time as a "line" of floating point numbers. The current version, TW 1.05, sends all the cancelled messages via sndmsg into the Tester. This is done for several reasons. For instance, the only input message at time "to" could be one we are cancelling.

The routine zip_forward (which is the code common to both new_end_of_vt and roll_it_back) views the input and output queues as queues of bundles. After a separate initialization, zip_forward cancels every message in the "next output bundle" until the "next input bundle's receive time" is as soon as the "next output bundle's send time". Indeed, we are just "bundle zapping" an output bundle with no corresponding input bundle.

The routine is called zip_forward because of the way it deals with self-propelled objects. Suppose object A just sends a message to object B at now and a message to itself at now + 20. If the event message M arrives for object A at time 50, then object A quickly generates messages at times 50, 70, 90, 110 and so on. If M's anti-message arrives, then go_forward would schedule A for 70. But zip_forward would cancel all the messages at 70, 90, 110 and so on, and schedule A for plus infinity. This routine doesn't often zap more than one or two output bundles, but a "zip" of length 20 has been observed.

Zip_forward works from two pointers pi and po which are the latest earlier messages in the input (pi) or output (po) queue which are "safe from side effects of the 'go forward' process". Usually o->ci becomes nxt_imsg_macro(pi). And the first output bundle cancelled (if any) will be the one that contains nxt_omsg_macro(po). Most of the work is to initialize pi and po while keeping our counters num_in(out)q_msg_past(future) current. There are of course some details missing in this discussion of zip_forward.

ODDS AND ENDS:

1. The routines sv_evtmsg, sv_qmsg and sv_qrmsg don't set the BLKPKT bit--i.e. sv_doit isn't called-- if svt < lvt.

2. The Ocblist is kept by lvt and not svt. Ocbcmp is the difference in lvts.