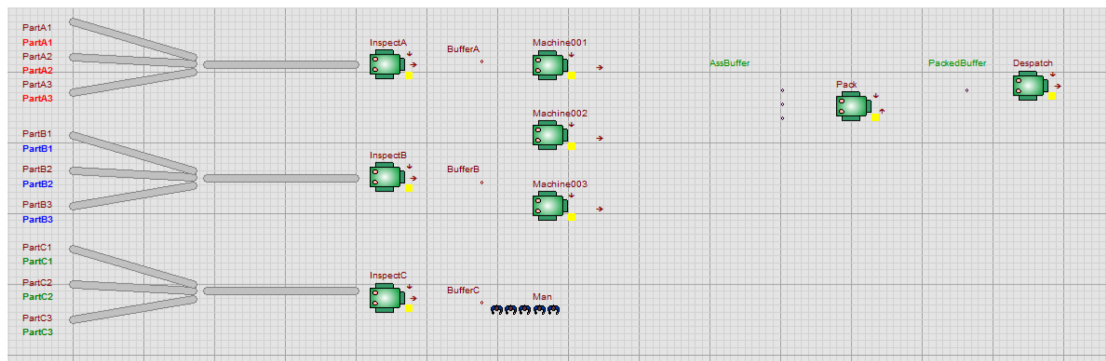
Speeding Up a Witness Model

Prepared by Steve Bridgman, Lanner Group - November 2008

Witness model performance is the result of a number of factors which influence the amount of processing that must be performed. Your model may have large numbers of elements processing many parts which will influence performance; it may have few elements but complex decision rules or have complex actions. There may be other delays caused by input and output routines or the computer environment.

The aim of this document is to assist you in developing the most efficient WITNESS models. It will use a scenario of modelling a production and packing facility to illustrate both the pit falls and best practice.

The planning manager of a company has built a WITNESS model of a proposed new production and packing facility. Parts A (Main Unit), B (Sub1), and C (Sub 2) are produced in three variants in a sister factory. They are initially fed by dedicated conveyors which merge and feed inspection stations. Inspected items are stored prior to assembly. The assembly process brings together one of each part of the same variant e.g. A2+B2+C2. Assembled units are stored prior to packing into customer orders of variable numbers, each of the three units. Once a load is available to ship, it is despatched.



The model is fairly simple, but is running quite slowly. A test set of orders has been created in a spreadsheet **Interface.xls** and this take 9 minutes to process. This will make running experiments very time-consuming, so how can this performance be improved?

The starting point is to record the time the model takes to run. The system function **GetActualTime** is very useful for this; it returns your computer system time, in units that you have set for your simulation, relative to the Clock Start Date set for your model. In our models the simulation units are minutes.

Use a real variable to store the system time in your Model/Initialize Actions

```
StartTime = GetActualTime ()
```

and print out the elapsed time in seconds at the end of your run – the stop is called at the end of packing the 1000th order.

```
IF CurrentOrder >= 1000
 PRINT (GetActualTime () - StartTime) * 60 ! The model time unit is minutes.
!
 STOP
```

```
ELSE
```

The model **BaseCase.mod** is run in batch mode. It runs in 534 seconds. As the basecase model runs you will notice that the time is updated every one time unit. This is a significant overhead on the system.

BaseCase.mod	No improvements made	534s	0s improvement

Go to **Model/Options/Run** and set the Batch Time Increment to 100. This is saved in **Batch100.mod**.

Batch100.mod	Batch Time Increment Increased	521s	13s improvement

The model reads orders and writes out completions, on an order by order basis, on **Actions on Finish** of the machine “Pack”. This may cause the model to slow down as it waits for Excel to read and write.

The model version **ExcelAtEnd.mod** reads the orders once into an array at the start of the run in Model Initialize and stores the results in an array which it then writes out at the end of the run.

The **Actions on Finish** of the pack machine become:

```
DIM row AS INTEGER
!
!row = CurrentOrder + 1
Packed (CurrentOrder) = TIME
IF CurrentOrder >= 1000
 XLWriteArray
("D:\\WitnessProjects\\WitnessSpeed\\Interface.xls.xls","Sheet1","$E2:E1001",Packed)
 PRINT (GetActualTime () - StartTime) * 60
!
 STOP
ELSE
 Bits = TotalBits (CurrentOrder)
 AttRegion = Region (CurrentOrder)
 CurrentOrder = CurrentOrder + 1
!row = CurrentOrder + 1 These are commented out as the orders are read into an array in
Model/Initialize
!XLReadArray ("D:\\WitnessProjects\\WitnessSpeed\\Interface.xls.xls","Sheet1","$A$" + row +
":$C$" + row,NumBits,1)
!XLReadArray ("D:\\WitnessProjects\\WitnessSpeed\\Interface.xls.xls","Sheet1","$D$" + row +
":$D$" + row,TotalBits,1)
!XLReadArray ("D:\\WitnessProjects\\WitnessSpeed\\Interface.xls.xls","Sheet1","$F$" + row +
":$F$" + row,Region,1)
ENDIF3220
```

Again this saves some time:

ExcelAtEnd.mod	Excel Read/Writes at End	515s	6s improvement

The base model has been written quickly and elements have been structured to clone them. This can be inefficient if it means processing strings into element names and vice versa. These rules call for string manipulation and also conversion to names on each execution of the rule.

Parts are pushed
PUSH to STR2NAME ("Conveyor" + RIGHTSTR (NAME2STR (**TYPE**),2))

And ConveyorA, ConveyorB and ConveyorC have pull rules
PULL from STR2NAME (LEFTSTR (NAME2STR (**ELEMENT**),9) + "1"),STR2NAME (LEFTSTR (NAME2STR (**ELEMENT**),9) + "2"),STR2NAME (LEFTSTR (NAME2STR (**ELEMENT**),9) + "3")

And the inspection machines have rules input rules:
PULL from STR2NAME ("Conveyor" + RIGHTSTR (LEFTSTR (NAME2STR (**ELEMENT**),8),1))

It may well be possible to use variables or group 0 attributes for these elements, so that they are processed and set only once, or the rules can be modified to make the references to elements explicit.

The model **NoManip.mod** has this change.

The push rule for PartA1 becomes:

PUSH to ConveyorA1 at Rear

ConveyorA has the input rule:
PULL from ConveyorA1 at Front,ConveyorA2 at Front,ConveyorA3 at Front

Inspection machine InspectA has input rule:
PULL from ConveyorA at Front

This change significant speeds-up the model.

NoManip.mod	Remove string manipulation	491s	24s improvement

Even after the last improvement, conveyors ConveyorA, ConveyorB and ConveyorC still Pull from conveyors ConveyorA1, A2 etc. Because these queuing type conveyors tend to run with space to load a part on the end, they are almost always looking for input.

The model version **ConPush.mod** has the input rules for ConveyorA, B and C set to Pull and the output from ConveyorA1, A2 etc set to Push.

This provides further significant improvement.

ConPush.mod	Push from conveyors	460s	31s improvement

Often in writing rules it is necessary to identify whether the contents of a buffer are sufficient to allow an operation to occur. For example, the Despatch machine must check if sufficient orders have been packed to make a viable load to a region. This uses a function **CanIDespatch**. The function loops around the PackedBuffer looking at each packed order counting the number of items in them.

```
IF NPARTS (PackedBuffer) = 0
 RETURN 0
ELSE
```

```
 IF SelectedQty > 0
  RETURN 1
 ENDIF
 FOR IVar = 1 TO 4
  CapUsed = 0
  SelectedQty = 0
  FOR JVar = 1 TO NPARTS (PackedBuffer)
   IF PackedBuffer AT JVar:AttRegion = IVar AND PackedBuffer AT JVar:Bits < DespCap -
CapUsed
    PackedBuffer AT JVar:Selected = 1
    CapUsed = CapUsed + PackedBuffer AT JVar:Bits
    SelectedQty = SelectedQty + 1
   ENDIF
  NEXT
  IF CapUsed >= DespMin
   RETURN IVar
  ELSE
! Not enough found so clear all selected flags
   FOR JVar = 1 TO NPARTS (PackedBuffer)
    PackedBuffer AT JVar:Selected = 0
   NEXT
   SelectedQty = 0
  ENDIF
 NEXT
ENDIF
RETURN 0
```

This rule repeatedly runs through the content of the buffer to assess the quantity ready for a region. It may be beneficial to have counters for the number of parts packed ready to despatch to each region and only attempt to build a load if there is enough.

The model **ExtraDespLogic.mod** has counter variables which are incremented on entry into buffer PackedBuffer and decremented on exit from packedbuffer.

The CanIDespatch function can then become:

```
IF NPARTS (PackedBuffer) = 0
 RETURN 0
ELSE
 IF SelectedQty > 0
  RETURN 1
 ENDIF
!
 FOR IVar = 1 TO 4
  CapUsed = 0
  SelectedQty = 0
  IF bitsinbuffer (IVar) >= DespMin
   FOR JVar = 1 TO NPARTS (PackedBuffer)
    IF PackedBuffer AT JVar:AttRegion = IVar AND PackedBuffer AT JVar:Bits < DespCap -
CapUsed
     PackedBuffer AT JVar:Selected = 1
     CapUsed = CapUsed + PackedBuffer AT JVar:Bits
     SelectedQty = SelectedQty + 1
    ENDIF
   NEXT
   IF CapUsed >= DespMin
    RETURN IVar
   ELSE
! Not enough found so clear all selected flags
```

```
    FOR JVar = 1 TO NPARTS (PackedBuffer)
     PackedBuffer AT JVar:Selected = 0
    NEXT
    SelectedQty = 0
   ENDIF
  ENDIF
 NEXT
ENDIF
RETURN 0
```

The test checks that there is enough packed before identifying which they are.

However, in this implementation this method does not improve performance. The conclusion from this is that, although this would appear to be a sensible speed-up, the overheads in collecting the data outweigh the benefits from using it.

ExtraDespLogic.mod	Check Buffer Contents	461s	-1s Improvement

One possible improvement is to replace Witness variables with Local Variables. This is done in **LocalVars.mod**. Again in this model, this change does not provide a measurable speed-up.

LocalVars.mod	Use Local Variables	461s	0s Improvement

In identifying how a model may be speeded-up you need to identify the elements that take most time to process. Often the clues are that a model slows down at particular times. This may because buffers are getting full and decisions are getting “harder” or may be that conditions mean that input rules are failing repeatedly. This may be evidenced by certain machines being empty; however are repeatedly trying to load.

Our model slows down dramatically after time 25000. If you run the model after this time you will note that the machines machine001, 002 and 003 are idle for significant periods and appear to be waiting for parts. The buffers that feed them have parts in them, but inspection shows the wrong mix of parts. Clearly this fact would trigger further analysis and review of the control logic.

Our next focus in speeding up the model is the **match rules** used by these machines.

```
MATCH/ANY
((PartA3 out of BufferA #(1) AND PartB3 out of BufferB #(1) AND PartC3 out of BufferC #(1))
OR (PartA1 out of BufferA #(1) AND PartB1 out of BufferB #(1) AND PartC1 out of BufferC
#(1))
OR (PartA2 out of BufferA #(1) AND PartB2 out of BufferB #(1) AND PartC2 out of BufferC
#(1))) AND Man #(1)
```

It is often useful to help the match rule by identifying conditions when it would succeed.

The match rule may have to scan through all members of the queue to check that a particular part is not there. If we can keep a simple count of the number of each type of part, then we only need to use match if all 3 counts are > 0. The variables **bitsinbuffer**A, B and C are used and these are incremented and decremented on input and output to the buffers.

The machine input rules become:

```
IF bitsinbufferA (2) > 0 AND bitsinbufferB (2) > 0 AND bitsinbufferC (2) > 0
 MATCH/ANY
PartA2 out of BufferA #(1) AND PartB2 out of BufferB #(1) AND PartC2 out of BufferC #(1)
AND Man #(1)
ELSEIF bitsinbufferA (3) > 0 AND bitsinbufferB (3) > 0 AND bitsinbufferC (3) > 0
 MATCH/ANY
PartA3 out of BufferA #(1) AND PartB3 out of BufferB #(1) AND PartC3 out of BufferC #(1)
AND Man #(1)
ELSEIF bitsinbufferA (1) > 0 AND bitsinbufferB (1) > 0 AND bitsinbufferC (1) > 0
 MATCH/ANY
PartA1 out of BufferA #(1) AND PartB1 out of BufferB #(1) AND PartC1 out of BufferC #(1)
AND Man #(1)
ELSE
 Wait
ENDIF
```

This improves run speed dramatically!

MatchPreCheck.mod	Help the match rule	8s	453s Improvement

This is an exceptional case of course. But, having reached this run speed, note that the smaller improvements we have made along the way become much more significant. They are largely independent – if you change the batch increment back to 1 the time for the run will nearly double. Putting back all the string manipulation would quadruple the time to run. So any speed improvement is worth having.

The final step in our search for speed is to look at the need for certain elements. The conveyors are modelled as **conveyors** which inherently have lots of events. In practice the key features of queuing conveyors are that they do not allow overtaking and that there is a minimum time that a part will take to move along them from rear to front. These features can be modelled using **delay buffers**.

The model **Buffered.mod** has this implemented.

Buffered.mod	Conveyors as Buffers	5s	3s Improvement

As you can see the model is now running 100 times faster than when we started. You won't be able to gain the same performance boost with every model. But the changes we've applied are worth considering for every model. They are summarised below:

1. Make sure the batch increment in your model is high enough so that batch running isn't slowed down.

2. Be careful with writing to and from Excel. It will be quicker to make fewer larger writes to Excel rather than lots of small writes to individual cells. If you can, then read all data at the start of the model run and write out all results at the end. Write out large arrays of data using XLWriteArray rather than looping around to write individual cells. If you are making many writes see if switching calculation in Excel to manual will speed things up. If there are lots of calculations based on data you write out it may be better if calculation is suspended, but remember to switch it back on after the model has finished.

3. Cut down string manipulation and conversion of strings to and from element names. If you have to construct strings related to machines can you store the values in attributes or variables so the conversions are done only once – not on every cycle.

4. Try to identify what is slow in the model. Use step mode and see if particular events take an appreciable time to process. Does the run speed change over time – what is different about the model state.

5. Try to setup data to help in decision making. Try to support Match rules or other rules that require looping round the content of buffers with counters to check if a selection is even feasible. These bounds type checks may prevent unnecessary processing.

6. Conveyors have lots of events – consider if buffers could be used in their place. Conveyors which pull have many cycles looking to load – consider pushing to the conveyor rather than pulling onto it.

7. Always structure rules so that the most selective test occurs first. Witness processes if conditionA and conditionB and conditionC as
   if conditionA then
     if conditionB then
       if conditionC then
   so only if conditionA is true will WITNESS evaluate conditionB – so make sure the one most likely to fail is first as a rule. (The exception would be if you have a simple condition with a low failure rate and a complex one with a high rate it may be better to do the simple one first to reduce the load on the complex one as much as possible – you are looking for "bang per buck"!)

8. Machines which fail to load will continue to attempt to load as other events occur. If its input rule takes a while to process, it may be worth loading a dummy part into it if it fails so that the machine doesn't keep trying to load (avoid comparing against TIME as this can lead to inefficiencies). This is sensible if the actual physical control system doesn't react in real time. For example, if an operator identifies no work waiting for a machine he won't be there waiting for work – but may come back later. Of course you will have to log these idle operations separately to account for them in statistics.