Causal Block Diagram algorithms

COMP 522 Fall Term 2002

Hans Vangheluwe

The following are concise notes on some simple algorithms needed to implement a Time Slicing simulator of Causal Block Diagrams. For more details, refer to your own class notes.

Causal Block Diagrams

A Causal Block Diagram model is a graph made up of connected operation blocks. The connections stand for signals. Blocks can be purely algebraic such as Adder and Product, or may involve some notion of time such as Delay, Integrator and Derivative. Furthermore, Input and Output blocks are often used to model the system's connection to its environment.

Though the actual Time Slicing simulation algorithm needs to know the function of the blocks in the block diagram, the order in which blocks need to be computed can be determined by abstracting the block diagram to a dependency graph (describing the dependencies between signals). The actual nature of the dependencies (possibly highly non-linear) is abstracted away.

The dependencies between signals on either side of a block which denotes some form of time delay (such as Delay, Integrator or Derivative), do not show up in a dependency graph as these blocks relate signal values at different time instants.

If the dependency graph does not contain dependency *cycles*, a simple *topological sort* will give an order in which the blocks need to be evaluated to give correct simulation results (*i.e.,* corresponding to the model's semantics). Note how there may be many different equivalent (from the point of view of the correctness of the simulation results) topological sort orderings corresponding to different orders in which neigbours of a node are visited.

Topological Sort

```
# topSort() and dfsLabelling() both refer
# to global counter dfsCounter which will be
# incremented during the topological sort.
# It will be used to assign an orderNumber to
# each node in the graph.
dfsCounter = 1

# topSort() performs a topological sort on
# a directed, possibly cyclic graph.

def topSort(graph):

  # Mark all nodes in the graph as un-visited
  for node in graph:
    node.visited = FALSE

  # Some topSort algorithms start from a "root" node
  # (defined as a node with in-degree = 0).
```

```
  # As we need to use topSort() on cyclic graphs (in our strongComp
  # algorithm), there may not exist such a "root" node.
  # We will keep starting a dfsLabelling() from any node in
  # the graph until all nodes have been visited.
  for node in graph:
    if not node.visited:
      dfsLabelling(node)

# dfsLabelling() does a depth-first traversal of a possibly
# cyclic directed graph. By marking nodes visited upon first
# encounter, we avoid infinite looping.

def dfsLabelling(node, graph):
  #  if the node has already been visited, the recursion stops here
  if not node.visited:

    # avoid infinite loops
    node.visited = TRUE

    # visit all neigbours first (depth first)
    for neigbour in node.out_neigbours:
      dfsLabelling(neighbour, graph)

    # label the node with the counter and
    # subsequently increment it
    node.orderNumber = dfsCounter
    dfsCounter += 1
```

Strongly Connected Components

If a CBD model's dependency graph contains dependency *cycles*, these need to be identified and replaced by an *implicit* solution (analytical or numerical). Note how often, a small Delay (or Integrator) is inserted to "break the loop" and hence avoid implicit solving. Finding dependency cycles is also known as locating *strongly connected components* in a graph. A strongly connected component is a set of nodes in a graph whereby each node is reachable from each other node in the strongly connected component.

```
# Produce a list of strong components.
# Strong components are given as lists of nodes.
# If a node is not in a cycle, it will be in a strong
# component with only itself as a member.

def strongComp(graph):

  # Do a topological ordering of nodes in the graph
  topSort(graph)

  # note how the ordering information is not lost
  # in subsequent processing and will be used during
  # Time Slicing simulation.
```

```
# Produce a new graph with all edges reversed.
rev_graph = reverse_edges(graph)

# Start with an empty list of strong components
strong_components = []

# Mark all nodes as not visited
# setting the stage for some form of dfs of rev_graph
for node in rev_graph:
  node.visited = FALSE

# As strong components are discovered and added to the
# strong_components list, they will be removed from rev_graph.
# The algorithm terminates when rev_graph is reduced to empty.
while rev_graph != empty:

  # Start from the highest numbered node in rev_graph
  # (the numbering is due to the "forward" topological sort
  # on graph
  start_node = highest_orderNumber(rev_graph)

  # Do a depth first search on rev_graph starting from
  # start_node, collecting all nodes visited.
  # This collection (a list) will be a strong component.
  # The  dfsCollect() is very similar to strongComp().
  # It also marks nodes as visited to avoid infinite loops.
  # Unlike strongComp(), it only collects nodes and does not number
  # them.
  component = dfsCollect(start_node, rev_graph)

  # Add the found strong component to the list of strong components.
  strong_components.append(component)

  # Remove the identified strong component (which may, in the limit,
  # consist of a single node).
  rev_graph.remove(component)
```

Time Slicing

A Time Slicing simulator for Causal Block Diagrams calculates the "behaviour" of a Causal Block Diagram model. In the case of no-time (data flow) and discrete-time CBDs, the result will be identical to the analytical result. In the case of continuous-time CBDs, the result approximates the analytical result. The error made in the approximation is determined by the simulation step-size (slice-size) Δt (given by the experimentation environment user). The simulator iteratively updates the variables in the system and advances the simulation time. The simulator has a main loop in which the state of each block is updated based on the state of its influencers.

Integrator/Delay/Derivative blocks are processed first, then all the Algebraic blocks (in sorted order). Note how there may be Integrator/Delay/Derivative block cycles. These would not have been detected during the strong component detection as we had removed all dependencies over Integrator/Delay/Derivative blocks (as they relate variables at different times and as such can't lead to zero-time-advance infinite loops). These Integrator/Delay/Derivative block cycles do imply that

we need to first store the results of processing these blocks in a temporary area until all have been processed and only then update their output !

After all blocks have been processed, the time is advanced by the time-increment. In the case of our Time Slicing algorithm for continuous-time, the time advance comes from the Euler discretization of the integrating blocks. The simulation loop will stop if some *termination condition* is reached. For numerical accuracy reasons, it may be necessary to set the time-advance Δt to a very small value.

The output (in file or on plot) the user wants may not require this resolution. Thus, the user gives a *Communication Interval* (CI). After every CI time-interval, output will be produced and not necessarily at every time slice !

Note that one can speed up the simulation dramatically by not processing blocks which are not needed to compute new values of state variables (*i.e.,* which are not needed as input to Integrator/Delay/Derivative blocks). These blocks are part of the *output function* λ rather than of the transition function δ (see the class on system specification) and need only be computed at communication intervals.

```
# Time Slicing simulation of a model

def timeSlicing(model):

  # Build a dependency graph for the model.
  # In practice, for efficiency reasons, no separate data structure is built.
  # Rather, the topSort() and strongComp() methods are
  # written such that they work on the original model graph.
  # Conceptually however, we reason about a separate dependency graph.
  dep_graph = dependency(model)

  # Identify strong components
  strong_components = strongComp(dep_graph)

  # Replace all "real" strong components
  for component in strong_components:
    if len(component) > 1:
      Replace this part of the model by an implicit solution
      This implicit solution is either obtained by a
      call to a numerical solver or by replacing the strong component
      by a new model-segment representing the analytical solution for
      that strong component.

      Note: we can also inform the user and cowardly give up
      (which is what you do in assignment 1).

  # After this pre-processing, we're ready for the actual
  # Time Slicing simulation.

  # Set the time to the initial time
  t = t_initial

  # The solution at t = t_initial

  Process ALL blocks in topsort order
```

```
Output (only) desired output variables at this time t

# Advance time to the next slice
t += delta_t (= 1 for discrete-time CBDs)

# The main simulation loop
# Often the termination condition is
#   (t > t_final)
while not termination_condition:

  Process time-delay blocks in any order
   # Caveat: as time-delay blocks may occur in cycles,
   # we need to store the results of processing these blocks
   # in a temporary area until all have been processed and only
   # then update their output !

  Process non-delay blocks in topSort order

  Output (only) desired output variables at time t

  # Advance the time
  t += delta_t (= 1 for discrete-time CBDs)
```