Causal Block Diagram algorithms

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)
```