Flow Network based Diagnostics for Incorrect Synchronous Models

Hans Olsson[1]

Dassault Systemes, Sweden, hans.olsson@3ds.com

Abstract

This paper will present a novel way to give diagnostics for incorrect synchronous models.

The goal is that this will ease the introduction of synchronous models, since unclear diagnostics often create a barrier for new users. In particular the case of separating the clocked and continuous parts will be considered, and shown to be equivalent to finding a "leak-flow" in a certain flow network, which can be solved using max-flow/min-cut techniques.

The result is efficient, easy-to-adapt, and gives diagnostics focused on correcting the issue.

We have not seen this idea used before in this context, even if in retrospect it seems natural and straightforward.

The methods have been implemented in Dymola 2019 (released in June 2018) and also in 3D Experience Platform 2019x.

Keywords: synchronous, graph theory, flow networks, minimal cut, error diagnostics

1 Introduction

Diagnostics for incorrect Modelica models is an important part of Modelica tools. Tools can implement advanced diagnostics, either by additional analysis based on the current Modelica language (Bonus and Fritzson, 2002), or in combination with adding restrictions to Modelica such as balanced models (Olsson *et al*; 2008).

After a short discussion about different forms of diagnostics for errors, we will start by introducing the synchronous part of Modelica 3.3, and then flow networks and the max-flow/min-cut theorem.

When presenting error diagnostics, important aspects include how early the diagnostics is given, and how localized the error is. The ideal situation is early detection and that at least one plausible correction is clearly located.

In particular, some diagnostics can be given as soon as the error is made, and tools can in those cases prevent the error from being introduced in the model, e.g., attempting to connect an electrical pin to a mechanical flange.

Other diagnostics can only be given when translating the complete model, e.g., missing a source signal in an expandable connector set.

An intermediate variant is those where we can give diagnostics for incomplete models without introducing false positives – i.e., we avoid diagnostics for issues that will naturally be corrected as part of completing the model; but it is still a global property.

The clock partitioning problem is one of these intermediate variants, which adds the restriction that the diagnostics should work on such incomplete models – in particular when equations are missing.

That also implies that we could present the diagnostics after each operation, but that is currently not implemented. The errors are not necessarily local – but it may still be that they could be corrected in one or a few places.

2 Synchronous Modelica

Modelica 3.3 added synchronous primitives (Elmqvist *et al*, 2012) intended to make it easier to model control systems that run on a sampled clock and connect to the continuous plant model. This section will only describe the concepts needed in this paper and is not a general introduction to synchronous modeling.

To illustrate we have a simplified model illustrating some of the concepts:

```
model FirstOne
  Real x,y,z;
equation
  when Clock(1) then
    2*x=sample(y);
  end when;
  when Clock() then
    z=x+1;
  end when;
  y=hold(z)+time;
end FirstOne;
```

The equation `2*x=sample(y);` is a clocked equation and only active when the corresponding clock ticks (every second as given by `Clock(1)`). Note that it is an actual equation – but only active when the clock ticks, in contrast to non-clocked when-clauses in Modelica which only allow a restricted form of equations where the left-hand side must be a variable.

One important aspect of the synchronous extension is that variables and equations are not declared to be continuous or clocked (in the example `x` and `z` are clocked and `y` is continuous), instead the clock-partition can be inferred using "clock inferencing".

Additionally, sub-models can be used in both the continuous and clocked domains; be restricted to one domain or the other, or connect the two domains in certain ways. That the same sub-model can be used in both domains imply that we cannot infer properties of the used models in general – but only infer properties for each specific component of those sub-models.

Specifically equations in `when Clock` must be clocked, and `sample` takes a continuous time input and returns a clocked value; and `hold` converts in the other way. It is not possible to directly use a clocked variable when it is not active, instead one must explicitly use `hold(z)` to get the last active value for `z`. Additionally `time` is always continuous time.

If the modeler makes mistakes, the clock inferencing may fail. A trivial example would be writing `y=z+time;` in the model above, since `time` must be continuous and `z` is given by a clocked equation. The algorithm in this paper gives diagnostics and recommended solution for such errors (mixing clocked and continuous) – and the algorithm is particularly suited for larger models where there are a large number of (potentially incorrect) intermediate steps in the inferencing.

In this example the clock for `z` is not specified, but automatically inferred to be the same as for `x`. The `Clock(1)` could also be duplicated, and it is then verified that the two clocks tick at the same time – ensuring that different clocks are not used accidentally. The example could also be written as:

```
model SimplerFirst
  Real x,y,z;
equation
  2*x=sample(y, Clock(1));
  z=x+1;
  y=hold(z)+time;
end SimplerFirst;
```

In this case we automatically infer that `x` and `z` are clocked, and `y` continuous time. The equation `z=x+1;` can on its own be either clocked or continuous time (and could in general be in a sub-model that works in both parts).

An advanced feature is that clocked equations may include differential equations provided a discretization method is specified for the clocked partition, e.g.:

```
model Discretized
  Real x,y,z;
equation
  2*x=sample(y, Clock(Clock(1),
        solverMethod="ExplicitEuler"));
  der(z)=x+1-z;
  y=hold(z);
end Discretized;
```

The solverMethod argument ensures that any differential equation in the partition (in this case `der(z)=x+1-z;`) will cause the corresponding variable, `z`, to be updated using one step of explicit Euler when the clock ticks. The `der`-operator cannot occur in clocked partitions without a deduced solverMethod (there are additional details regarding different partitions that are not relevant for the analysis in this paper).

A final important aspect is that Modelica supports graphically connecting components – continuous, clocked, and even components mixing the two domains. There are also libraries of models, including Modelica_Synchronous that contain tested standard models.

3 Flow networks

A flow network (Ford and Fulkerson, 1956; or any general overview such as Cormen *et al,* 1993); is a directed graph where each edge has an arbitrary nonnegative capacity. When modifying the graph the capacities can become zero, and in that case we view it is as if the edge is not present.

A flow in such a network satisfies a number of constraints, in particular the flow in each edge does not exceed its capacity and except for source and sink vertices the in-flow to a vertex matches the out-flow from that vertex. In Figure 1 a small flow network with flows is shown, the source is marked with "s" and the sink with "t" (for target) and each edge has two numbers, the first is the current flow and the second is the capacity. The edges where the current flow equals the capacity are saturated.

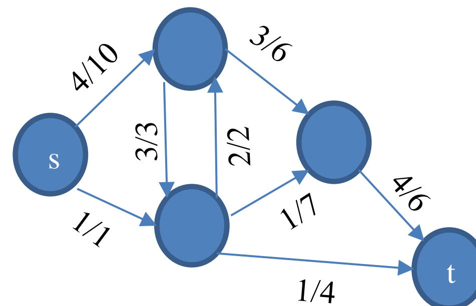

Figure 1 A small flow network.

Without loss of generality we can assume that there is only one source and one sink (Ford and Fulkerson, 1956). If there are e.g., multiple sources it is known that we can introduce a "super-source" with edges of infinite capacity going to each source, and treating those original sources as normal vertices; unless there are additional restrictions on the flows.

3.1 Minimal cut theorem

A **disconnecting set** of edges partitions the vertices into two sets – one containing the source and another the

sink. A disconnecting set without redundant elements is a cut.

The max-flow min-cut theorem, also known as "minimal cut theorem" (Ford and Fulkerson, 1956); states that the maximal flow obtainable in a network is the minimum of the sum of capacities of the edges in the set taken over all disconnecting sets. (Note: even if it is the minimum for all disconnecting sets the minimum is clearly for a disconnecting set without redundant elements, i.e., for a cut.)

If we revisit the previous small flow network we see that the maximum flow is 10, and the minimal cut is shown in red in Figure 2; and the other edges as dotted.

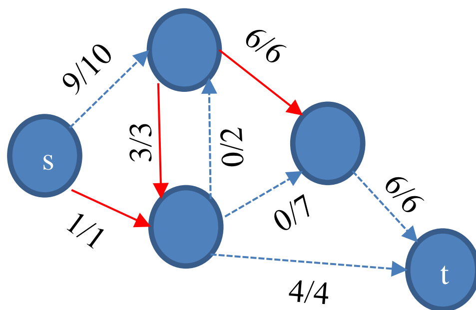

Figure 2 Minimal cut

Note that the minimal cut is not necessarily unique – another option would be to replace the 6/6 edge with the other 6/6 edge, and a third option would be the two edges going into "t". The two edges going from the source are clearly a cut, but its total capacity is 11 and it is therefore not a minimal cut. The red 6/6 edge and the 4/4 have a sum of capacities of 10, but is not a disconnecting set and thus not a cut. The red edges in union with the 4/4 edge form a disconnecting set that is neither a cut nor minimal.

There exists a number of algorithms for constructing the minimal cut and the maximum flow, with different running time in terms of number of edges and vertices; (Cormen *et al,* 1993).

3.2 Augmentation path

If a flow network allows a flow between the source and the sink we can find a path – called chain of edges in (Ford and Fulkerson, 1956) connecting them. The maximum flow through that path is the minimum capacity of any of the edges in the path.

After "subtracting" this flow from the graph one can attempt to find an additional path (called "augmentation path") connecting the source and the sink, and repeating this leads to the algorithm called Ford-Fulkerson based on (Ford and Fulkerson, 1956).

Subtracting the flow means both reducing the capacities of the used edges, and adding a capacity in the reverse direction; the latter is needed since we will sometimes later reduce the flow through specific vertices.

The path will later be used for error diagnostics, and thus redundant edges will cause a problem in at least two cases:

If the graph has cycles a vertex could appear multiple times in the path, but that can only decrease the flow through the path and the algorithm thus avoids revisiting vertices.

Additionally, if there are multiple sources a path could start at one source and then have an edge leading to a different source (and similarly for sinks). By treating all sources as visited at the start and avoiding revisiting vertices that is avoided for the sources, and by stopping at the first sink reached it is avoided for sinks.

The current implementation does not use a breadth-first search for the path, but that would naturally avoid the previous issues.

A major restriction of the algorithm is that this only converges in a finite number of steps if the capacities are integers (or in general rational numbers); and has a running time of O(number of edges*maximum flow). This follows from the fact that we can find one augmentation path in running time proportional to the number of edges, and each augmentation path has a flow of at least one.

We currently do not use any specific heuristic for finding the augmentation path, but a breadth-first search is generally a good heuristic avoiding specific problems for large maximum flow (Cormen *et al,* 1993).

Assuming the maximum flow is small this simple algorithm compares favorably to recent algorithms; that instead are superior if the maximum flow is large or the capacities are real numbers; as their running time only depend on the number of edges and vertices.

4 Min-cut and Clock partition

We will now combine the clock partition and the flow network.

4.1 Flow networks for synchronous models

Based on a model with synchronous parts we can construct a flow network where the variables and equations that must be continuous are sources, and variables and equations that must be clocked are sinks.

Both equations and variables are vertices in this graph, and edges connect equations to variables appearing in the equation – unless the variables appear inside certain primitives, in this paper we will only discuss `sample` and `hold`, but in general it also includes "Clock with Boolean condition". The variables inside these primitives are instead sources or sinks. The edges are also added in the opposite direction (with the same capacity in both directions) so that we get a symmetric directed graph. Alternatively, we can replace this pair of edges with one bidirectional edge with capacities in both directions – initially equal.

Having edges in both directions implies that there are cycles in the graph, but the chosen algorithm can handle that.

The capacities for all edges can be selected as positive integers; the exact values will be discussed later.

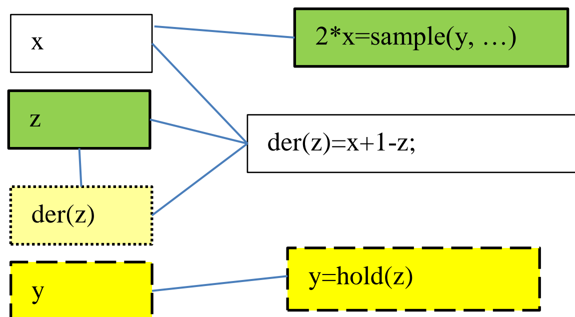

Figure 3 Flow network for the model "Discretized"

In Figure 3 we see the flow network corresponding to the previous model "Discretized" illustrating most of the concepts, where continuous equations and variables (argument of `sample`) are marked in yellow (and dashed outline) and clocked equations and variables (argument of `hold`) are marked in green. The `der(z)`-variable is special and marked in lighter yellow (and dotted outline), indicating that without a solverMethod it must be in a continuous partition (which would cause a leak-flow between the partitions). There is also an edge from `der(z)` to `z` indicating that they should be in the same partition (in later graphs derivative-variables will not be separate nodes). Since there is a solverMethod attached to the clock-partition, `der(z)` is just a normal variable and there is no leak-flow.

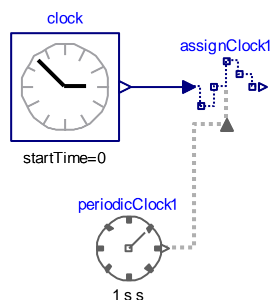

Figure 4 Incorrect assignClock

In Figure 4 we have an incorrect model from MCP-0030 (Frenkel, 2018). The corresponding flow network is shown in Figure 5, where the continuous part (due to `time`) is marked in yellow and the clocked part (due to `when Clock()`) in green, and the edges that are not part of the augmentation path are dotted. The arrows on the edges indicate the direction of the flow.

The saturated edges (i.e., potential minimal cuts) are shown in red and wider. It is common that the saturated edges occur in pairs and the implementation handles that, but we will in the future investigate alternative formulations that avoid this.

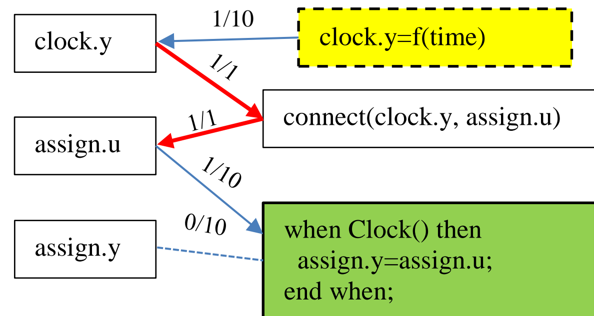

Figure 5 Synchronous simple flow network

The graph is bipartite with equations and variables forming the two parts as is normal in Modelica, but since both sources and sinks can appear in both parts this fact does not seem useful for analyzing this flow network. This implies that the edges in the cut can go both from equation to variable and vice-versa.

Note that the problem of assigning variables to equations in Modelica is equivalent to solving a maximum-flow problem on such a bipartite graph.

4.2 The significance of the flow

A correct model can be partitioned into zero or more clocked parts, and zero or more continuous parts. This corresponds to separating the graph into separate parts, and thus a zero flow.

If the flow is positive it indicates that graph cannot be partitioned in this way and the flow gives "leakage" between continuous and clocked parts. If there are multiple disjoint errors there will be multiple "leakages" increasing the flow; i.e., a higher flow can be seen as an indication of a more incorrect model.

The cut indicates which variables to remove from the equations to restore the partition. Replacing the variables by `sample()` or `hold()` variants of the same variables removes the edge, without excessively altering the model structure.

Similarly, the corresponding path(s) between source and sink is important, since that allows the user to see that there is an unwanted path between the clocked and continuous parts.

4.3 The capacities of edges

The previous method would work for any set of positive integers as edge-capacities, and give some cut between clocked and continuous parts.

Good diagnostics for errors can thus be seen as finding a suitable heuristic for the capacities. The basic idea is that we give high capacity to edges that we do not want in the cut-set; or roughly that high capacity corresponds to high trust in that equation.

As a first attempt, we chose capacity 1 for edges corresponding to connection-equations, and 10 for other edges. In the future, we are considering having higher weights for equations from tested libraries.

DOI
10.3384/ecp19157743

4.4 Algorithm

The following presents pseudo-code outlining the algorithm. Note that similarly as (Ford and Fulkerson, 1956) it is not a completely specified algorithm as there are multiple ways of finding the augmentation path.

Additionally, the source-cut part can use any algorithm that finds reachable nodes in a graph.

```
Sources={time}
Targets={}
Edges={}

// Build graph based on equations:
for eq in Equations loop
  // Low capacity for likely errors
  cap=if eq is connection then 1 else 10;
  for var in Incidence(eq) loop
    if var inside sample then
      Sources+={var};
      eq.isClocked=true;
    elseif var inside hold then
      Targets+={var};
      eq.isNonClocked=true;
    else
      // Edge(from->to, cap)
      Edges+={Edge(eq->var, cap)};
      Edges+={Edge(var->eq, cap)};
    end if;
  end for;
  if eq.isClocked then
   Targets+={eq};
  end if;
  if eq.isNonClocked then
   Sources+={eq};
  end if;
end for;

// Ford-Fulkerson finding max-flow:
maxFlow=0;
loop  // Find path of edges having cap>0
  augmentPath=FindPath(Sources, Targets);
  if augmentPath=={} then
    break;
  end if;
  // Possible flow for path
  flow=min(e.cap for e in augmentPath);
  maxFlow+=flow;
  // Subtract augmentPath flow:
  for e in augmentPath loop
    e.cap-=flow;
    reverse(e).capacity+=flow;
  end for;
end loop;

// Find source-cut, i.e. the cut
// closest to the source

// We first find all vertices
// reachable from the sources
SourceConnected={};
AddTo=Sources;
while not AddTo.empty() loop
  vertex=AddTo.front();AddTo.pop_front();
  if not vertex in SourceConnected then
    for e in Edges.from(v) loop
      if e.cap>0 then
        AddTo.push_back(e.target);
      end if;
    end for;
  end if;
end while;

// Find all edges with 0 capacity
// that leaves this set
SourceCut={};
for v in SourceConnected loop
  for e in Edges.from(v) loop
    if e.cap==0 and
      not (e.to in SourceConnected) then
      SourceCut+={e};
    end if;
  end for;
end for;
// And similarly for the target-cut
```

5 Examples

MCP-0030 (Frenkel, 2018) was made to solve the same problem as this paper. It contains one example of a bad model – shown in Figure 4. The rationale for the MCP was that earlier diagnostics just listed all equations and variables that failed for clock inference and it was not helpful for users. Note that the ideas presented here were implemented before the MCP.

With the approach in this paper, this example gives the diagnostics:

Continuous time parts and discrete parts don't decompose.
It is necessary to introduce sample or hold elements replacing:
connect(clock.y, assignClock1.u);
The following sequence indicates that the involved variables and equations are continuous time:
clock.y :
clock.y = clock.offset+(if time < clock.startTime then 0 else time- clock.startTime);
However, this is in contradiction with
assignClock1.u:
when assignClock1.clock then

```
    assignClock1.y = assignClock1.u;
end when;
```

The two sequences of equations and variables are constructed from the augmentation paths, cut according to the minimal cut.

It could be even clearer by showing that clock.y is continuous due to using time, and in this case the connection between clock.y and assignClock1.y was the obvious culprit.

We will now consider variations of an example from Modelica_Synchronous shown in Figure 6.

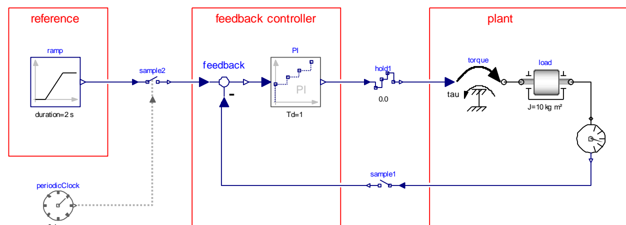

Figure 6 Textbook controller

If we had forgotten one of the synchronous primitives (there are two sample-blocks and one hold-block) the goal would be that diagnostics would recommend adding it.

The possible errors will be listed starting from the one ones that are simplest to investigate, and then proceed to the more complicated ones.

Without sample2 the diagnostics will pinpoint that:

```
It is necessary to introduce sample or hold elements
replacing:
    connect(ramp.y, feedback.u1);
```

If sample2 were replaced by a gain-block the diagnostics would be:

```
Continuous time parts and discrete parts don't
decompose.
It is necessary to introduce sample or hold elements
replacing:
    connect(ramp.y, gain.u);
or:
    connect(gain.y, feedback.u1);
or some variation of this.
```

The two proposed corrections are two different min-cuts (one close to source, one to sink). In general there could be a large number of possible min-cuts, and listing them all could be time-consuming and unlikely to help users.

If both sample1 and sample2 are missing, the diagnostics state:

```
It is necessary to introduce sample or hold elements
replacing:
    connect(feedback.y, PI.u);
```

This shows an interesting change, since the proposed change moves the feedback-component from the clocked part to the continuous part.

If only sample1 was removed the method described so far would see two possibilities:

```
    connect(torque.tau, hold1.y);
    connect(speed.w, feedback.u2);
```

The first suggestion has two problems: firstly, it would make more sense to remove the hold-block than introduce a sample-block, but secondly and more importantly, that model would not translate, since the der-operator is used in a clocked partition without solverMethod.

That is handled by running the algorithm on a slightly different flow network, which is constructed by considering the der-operator part of the continuous partition. That results in:

```
Continuous time parts and discrete parts don't
decompose, when there is no solverMethod attached to
the clock.
It is necessary to introduce sample or hold elements
replacing:
    connect(speed.w, feedback.u2);
```

Without hold1 we get:

```
Continuous time parts and discrete parts don't
decompose, when there is no solverMethod attached to
the clock.
It is necessary to introduce sample or hold elements
replacing:
    connect(torque.tau, PI.y);
```

Indicating that the correction is instead to introduce hold1.

If both sample1 and hold1 are missing the model is valid for check, but translation would give (flow network in Figure 7):

```
Continuous time parts and discrete parts don't
decompose, when there is no solverMethod attached to
the clock.
It is necessary to introduce sample or hold elements
replacing:
    connect(speed.w, feedback.u2); connect(torque.tau,
PI.y);
    The following sequence indicates that the involved
variables and equations are continuous time:
    speed.w : speed.w = der(speed.flange.phi);
    torque.tau : torque.flange.tau = -torque.tau;
    torque.flange.tau :
load.flange_a.tau+torque.flange.tau = 0.0;
    load.flange_a.tau : load.J*load.a =
load.flange_a.tau+load.flange_b.tau;
    load.a : load.a = der(load.w);
    However, this is in contradiction with
    feedback.u2 : feedback.y = feedback.u1-
feedback.u2;
    feedback.u1 : sample2.y = feedback.u1;
    sample2.y : sample2.y = sample(sample2.u,
sample2.clock);
    feedback.y : feedback.y = PI.u;
    PI.u : when Clock_0 then
      PI.x = previous(PI.x)+PI.u/PI.Td;
```


DOI 10.3384/ecp19157743

PI.y = PI.kd*(PI.x+PI.u);
end when;
The sub clock, BaseClock_0.SubClock_1, includes derivatives, but no solver method is specified.

The last line indicates that one way of correcting the model is to specify a solverMethod for the partition. In that case the plant-part will be discretized as part of the clocked partition.

However, the first part of the error message indicates that another solution is to introduce sample and hold. This error message lists two connect-statements, and the users has to replace both of them.

The part "The following sequence…" would as default be collapsed since it is quite long and only shows what is already stated. However, even if lengthy it still only lists relevant variables and equations, e.g. load.flange_a.phi and load_flange_b.phi are also part of the continuous-time partition – but they are not part of any augmentation path used and thus not included.

6 Implications for models

The examples demonstrate that no changes are needed to support these diagnostics. However, changes in models and/or the language can still be helpful to improve the diagnostics further.

In particular, models representing external controllers can currently be written as

```
model Controller
  extends SI2SO;
equation
  y=do_step(u1, u2);
end Controller;
```

Here do_step is an external C function (possibly part of an FMU), and each do_step updates the internal state and thus it should be run at every sampling point of the inferred clock.

That works in correct models where it is assigned to the clocked part and run at every sampling point. However, if the controller is incorrectly connected that can end up in the continuous part; which is not intended.

Modelica 3.4 (Olsson (editor), 2017) has restrictions for impure functions, but if `do_step` is not declared as impure that will not generate diagnostics. For the future we might consider treating impure function as sinks in the graph.

One possibility is to change the model to:

```
model Controller
  extends SI2SO;
equation
  when Clock() then
    y=do_step(u1, u2);
  end when;
end Controller;
```

That variant works and ensures that it is part of a clocked partition, providing better diagnostics – but it looks slightly distracting. A future possibility could be to introduce a form of "Clocked model" where all equations are seen as clocked.

7 Future work

For the future there are multiple lines of potential work – one is improving the current work by improving the diagnostics, another is using this for unrelated problems.

7.1 Other uses of min-cut

An obvious question is whether the same concept can be applied to other problems.

The characteristics leading to min-cut being a good fit for this problem are:

- Vertices can be partitioned into two parts where certain vertices (the sources and sinks) must be in certain partitions.
- Corrections correspond to removing edges.

If we consider the separation of variables into different clocks (and similarly for sub-clocks) we see that they sort of match the first, but not the second criteria:

- One Clock could be a selected as a source and another Clock-variables as sink. This works if there are two Clocks mixed together that should not be mixed, but if there are three or more Clocks mixed together this is not ideal (but at least it produces some diagnostics).
- However, removing an edge is not the only possible correction – another possibility is that the Clocks should be the same.

Clock-partitioning diagnostics should thus both include the possibility of separating the graphs, and also changing the Clocks to be the same, i.e., we can view it as two distinct cases (like for missing solverMethod or missing sample and hold).

For sub-clocks, the possibility of merging the clocks is even more complicated, since they can depend on multiple sub-clock factors.

Unrelated to synchronous models we believe this kind of diagnostic can be useful when breaking dependencies using `decouple` to allow parallelization – as described in (Elmqvist *et al*, 2014). It cannot directly help with `decouple` failing to split the system of equations into smaller part, but it can detect that the two sides of decouple are connected in unexpected ways, which has a tendency to occur.

7.2 Implementation

The algorithm was originally implemented Dymola 2019 (released in June 2018) and also in 3D Experience Platform 2019x. The handling related to solverMethod will be added in Dymola 2020, and all diagnostics are from that version.

Acknowledgements

The incorrect models from various users were one important aspect for starting this work; another inspiration to start the work was an effort to understand the existing algorithms for synchronous models developed by Sven Erik Mattsson (Elmqvist *et al*, 2012).

The proposal for a new primitive in MCP-0030 (Frenkel, 2018) gave a major inspiration to describe the approach.

Feedback from my colleagues and reviewers were helpful in making this paper clearer.

References

Peter Bunus, and Peter Fritzson (2002): Methods for Structural Analysis and Debugging of Modelica models. *Proceedings of the 2nd International Modelica Conference.* 157-165.

Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest (1993): Introduction to Algorithms.

Hilding Elmqvist, Martin Otter, Sven Erik Mattsson (2012): Fundamentals of Synchronous Control in Modelica. *Proceedings of the 9th International Modelica Conference.* 15-26. doi: 10.3384/ecp1207615

Hilding Elmqvist, Sven Erik Mattsson, Hans Olsson (2014): Parallel Model Execution on Many Cores. *Proceedings of the 10th International Modelica Conference*. 363-370. doi: 10.3384/ECP14096363.

L. R. Ford, Jr and D. R. Fulkerson (1956): Maximal Flow Through a Network. *Canadian Journal of Mathematics* 8:399-404. doi:10.4153/CJM-1956-045-5.

Jens Frenkel (2018): Modelica Change Proposal MCP-0030 IsClocked Operator.

Hans Olsson, Martin Otter, Sven Erik Mattsson, Hilding Elmqvist (2008): Balanced Models in Modelica 3.0 for Increased Model Quality, *Proceedings of 6th International Modelica Conference*, vol. 1:21-33.

Hans Olsson (editor) (2017): Modelica A Unified Object-Oriented Language for Systems Modeling Language Specification Version 3.4.

8 Appendix

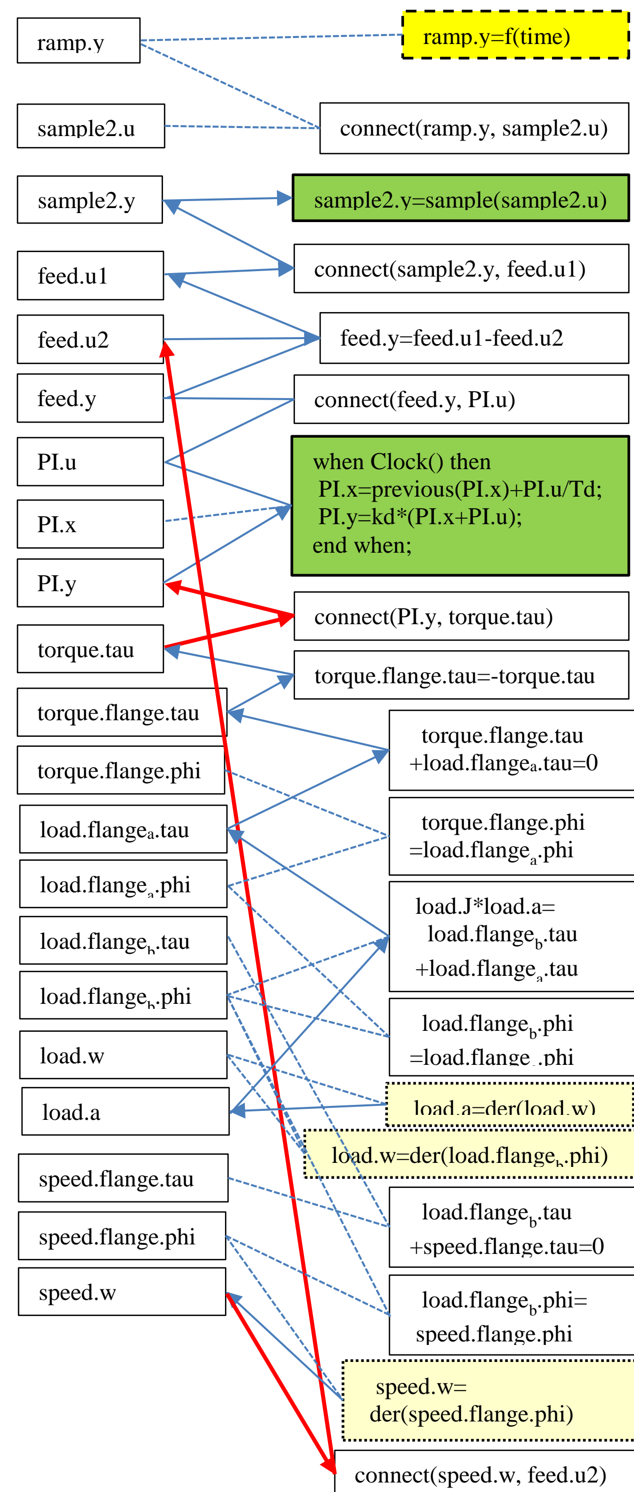

Figure 7 Flow network for advanced example

DOI
10.3384/ecp19157743