Modeling, Identification and Control, Vol. 36, No. 1, 2015, pp. 53–65, ISSN 1890–1328

Structural Observability Analysis of Large Scale Systems Using Modelica and Python

M. Anushka S. Perera Bernt Lie Carlos Fernando Pfeiffer

Telemark University College, Kjølnes ring 56, P.O. Box 203, N-3901 Porsgrunn, Norway. E-mail: {from,Bernt.Lie,Carlos.Pfeiffer}@hit.no

Abstract

State observability of dynamic systems is a notion which determines how well the states can be inferred from input-output data. For small-scale systems, observability analysis can be done manually, while for large-scale systems an automated systematic approach is advantageous. Here we present an approach based on the concept of structural observability analysis, using graph theory. This approach can be automated and applied to large-scale, complex dynamic systems modeled using Modelica. Modelica models are imported into Python via the JModelica.org-CasADi interface, and the Python packages NetworkX (for graph-theoretic analysis) and PyGraphviz (for graph layout and visualization) are used to analyze the structural observability of the systems. The method is demonstrated with a Modelica model created for the Copper production plant at Glencore Nikkelverk, Kristiansand, Norway. The Copper plant model has 39 states, 11 disturbances and 5 uncertain parameters. The possibility of estimating disturbances and parameters in addition to estimating the states are also discussed from the graph-theory point of view. All the software tools used on the analysis are freely available.

Keywords: structural observability, Modelica, large-scale systems, CasADi, Python, graph-theory, JModelica.org, NetworkX, PyGraphviz

1. Introduction

Knowing the internal state of a dynamic system is important in many applications such as state feedback. However, measuring all state variables is usually impossible or impractical. What is more realistic is to estimate the state variables based on a finite set of measurements. The notion of observability characterizes whether a given set of measurements is adequate to estimate the state of the system. For linear time invariant systems, if the rank of the observability matrix is equal to the dimension of the state space, then the system is observable Simon (2006). For nonlinear dynamic systems diverse local observability definitions can be considered, for example using Lie derivatives Liu et al. (2012). In addition to analyzing observability for a given set of measurements, it would also be useful (especially for large-scale complex systems) to systematically find the minimum set of measurements which makes the system observable. By exploiting the model structure (algebraic dependencies among state and output variables), we can infer the minimum number of measurements and the possible choices to select from. Structural (or algebraic) observability is a fundamental property that provides a necessary condition for observability, and often it may also be sufficient for many systems Reinschke (1988), Liu et al. (2012). Structural observability analysis can be done using graph-theoretic techniques. Under some assumptions, unknown disturbances/parameters can be estimated (for example using Kalman filtering techniques Simon (2006)) by augmenting the system with them as state variables, making it necessary to check the observability of the augmented system. Struc-

doi:10.4173/mic.2015.1.4 © 2015 Norwegian Society of Automatic Control

tural observability analysis via graph theory offers a visual means to pinpoint measurements needed to estimate states/disturbances/parameters, or to detect which cannot be estimated at all in the augmented system.

JModelica.org is a Modelica-based simulation tool that makes possible to make Modelica models available as symbolic model objects in Python with the help of the JModelica.org-CasADi interface. The symbolic models can then be used in structural analysis using Networkx, Pygraphviz, and Python packages.[1]

This paper demonstrates the usefulness of using Python and relevant Python packages in analyzing structural observability for large-scale systems via graph theory. As a case study, the Copper production plant at Glencore Nikkelverk AS, Kristiansand, Norway is considered. The Copper plant model contains 39 states, 11 disturbances, and 5 uncertain parameters. The possibility of estimating disturbances and parameters additionally to the states will be discussed in a graph-theocratic point of view.

Section 2 gives a basic description about graph-theoretic concepts which are needed in the subsequent sections. Section 3 discusses structural observability in graph-theoretic point of view. A way of automating structural observability analysis in Python is given in Section 4. A demonstration of our development is done based on a real process and it is given in Section 5.

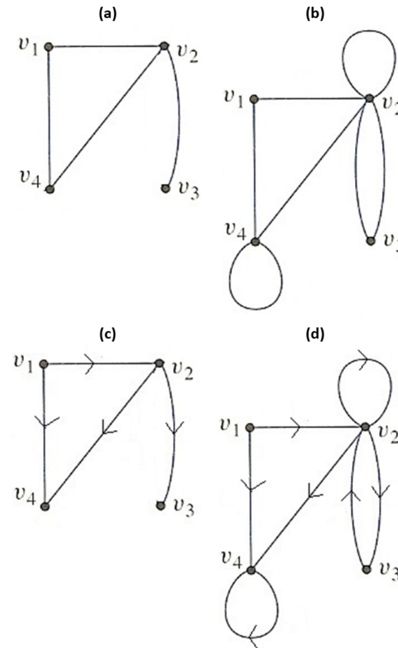

Figure 1: (a) Undirected and without self-cycles/loops and multiple edges. (b) Undirected and with self-cylcles/loops and multiple edges. (c) Directed and without self-cycles/loops and multiple edges. (d) A directed and with self-cycles/loops and multiple edges.

2. Graph-theoretic concepts

A graph G is denoted by $G = (V, E)$ where V is a set of nodes (or vertices) and E is a set of edges.[2] An edge connects two nodes v_i and v_j (where $v_i, v_j \in V$) and denoted by (v_i, v_j). The node v_i and v_j are incident to (v_i, v_j) and v_i and v_j are said to be adjacent nodes. A graph may be directed or undirected. In undirected graphs, edges are marked with directed lines while in undirected graphs it is not. For undirected graphs, (v_i, v_j) and (v_j, v_i) are identical. A directed/undirected graph may allow multiple edges among nodes. In short, digraph stands for directed graph. Let an edge $e_i = (v_i, v_j) \in E$, then v_i is the initial-vertex and v_j is the final-vertex. As a shorthand notation for a directed edge, let $(v_i, v_j) \equiv v_i \rightarrow v_j$.

A path is a sequence of edges connected one after another. A path has an initial node and a final node. Number of edges in a path is called the length of it. If a path contains no node appearing more than once, then it is a simple path. A path where the initial and final nodes are identical is called a close path . A cycle is a closed path with no node appearing more than once except the initial and the final nodes. Cycles with length 1 are self-cycles/loops. A set of cycles such that no two cycles have at least once common nodes are called a cycle family. A cycle family which covers all the nodes in a graph is called a spanning cycle family. v_i and v_j are strongly connected if paths from v_i to v_j and from v_j to v_i exist. A strongly connected component (SCC) is a sub-graph (of a directed graph) where each vertex in SCC is strongly connected with all other vertices in SCC. A digraph is said to be strongly connected if any two nodes in it are strongly connected. See figures 1 for several examples. Consider figure 1-d. $\{(v_1, v_2), (v_2, v_4), (v_4, v_4)\}$ is a path and its length is 3. $\{(v_1, v_2), (v_2, v_3)\}$ is a simple path. $\{(v_2, v_3), (v_3, v_2)\}$ is a closed path and it is a cycle as well. $\{(v_2, v_2)\}$ and $\{(v_4, v_4)\}$ are self-cycles. Nodes v_2 and v_3 are strongly connected. $\{\{(v_2, v_2)\}, \{(v_4, v_4)\}\}$ and $\{\{(v_4, v_4)\}, \{(v_2, v_3), (v_3, v_2)\}\}$ are two cycle families.

Instead of using the term "simple path", we use "ele-

[1]Alternatively, it is possible to create symbolic mathematical models using the Python package SymPy which is a CAS — CAS stands for Computer Algebra System — tool and then use Networkx and PyGrapViz. However, this method is more limited since it does not support the modeling power available in Modelica.

[2]Refer Bondy and Murty (2008) for graph theory.

mentary path” for digraphs. Similarly, instead of “simple cycle”, “elementary cycle” is used. A stem is an elementary path. The initial and final nodes are respectively called the “root” and the “top” of the stem. A root is also called a driver node. A bud is an elementary cycle with an additional directed edge where its final node is one of the nodes in the cycle. This additional edge is called the distinguished edge of the bud. A directed cactus is made out of a stem and buds connected in a special way. The initial node of the distinguished edge of a bud is connected to any node in the stem except the top or it may be connected to a node of another bud. See figure 2. A cactus has a driver node which is the root of the stem in the cactus. If there are vertex disjoint cacti covering all nodes in a given digraph, then they are called spanning cacti and cacti have more than one driver nodes.[3] See supplementary information to Liu et al. (2011)) for further details.

Consider a subset of edges M in an undirected graph where no two edges share common nodes. Nodes in M are said to be matched. M is a maximum matching if there exists no edge set M' such that $|M'| > |M|$.[4] M is perfect if each node in the graph is in M. See figure 3 and note that thick color lines are matched edges. A path with edges alternating between $E \setminus M$[5] and M is an M-alternating path. M-alternating path is M-alternating path with odd number of edges where the starting and the final edges are not in M. According to figure 3, $\{(v_4, v_3), (v_3, v_8), (v_8, v_1), (v_1, v_7), (v_7, v_6)\}$ is an M-augmenting path. For digraphs, a matching is a subset of edges where no two edges share common nodes and a node is said to be matched if that node is the ending node of a matched edge Liu et al. (2011). See figure 4 where $M = \{(v_6, v_7), (v_1, v_8), (v_3, v_4)\}$ and v_7, v_8, and v_4 are matched nodes.

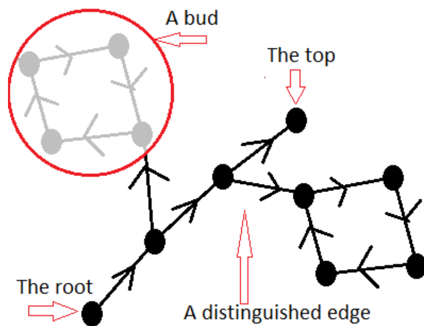

Figure 2: A cactus with two buds.

By formulating a bipartite graph (in short a bigraph)

[3] The plural of “cactus” is “cacti”.
[4] $|M|$ is the cardinality of M.
[5] $E \setminus M = \{e | e \in E \; \& \; e \notin M\}$.

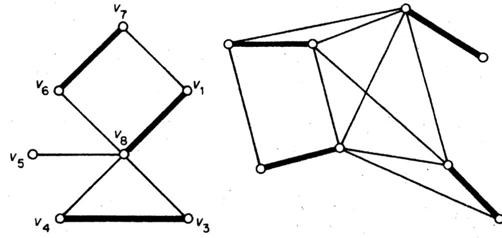

Figure 3: To the left - a maximum matching ($M = \{(v_4, v_3), (v_8, v_1), (v_6, v_7)\}$). To the right - a perfect matching.

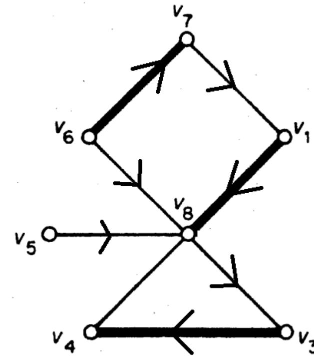

Figure 4: A matching for a digraph.

appropriately, a maximum matching for digraphs can be efficiently found. In a bipartite, there are two disjoint sets of nodes V_A and V_B such that edges only exist between V_A and V_B. See figure 5.

3. Structural observability

Consider the linear time invariant (LTI) state space model

$$\begin{aligned} \dot{x} &= A \cdot x + B \cdot u, \\ y &= C \cdot x + D \cdot u, \end{aligned} \tag{1}$$

where $A \in \mathbb{R}^{n_x \times n_x}$, $B \in \mathbb{R}^{n_x \times n_u}$, $C \in \mathbb{R}^{n_y \times n_x}$, $D \in \mathbb{R}^{n_y \times n_u}$, $x = [x_1, x_2, \ldots, x_{n_x}]^T$, $u = [u_1, u_2, \ldots, u_{n_u}]^T$, and $y = [y_1, y_2, \ldots, y_{n_y}]^T$. Once the output vector $y \in \mathbb{R}^{n_y}$ and the input vector $u \in \mathbb{R}^{n_u}$ are known, then the state vector $x \in \mathbb{R}^{n_x}$ can be estimated if the system is observable. Analyzing observability based on the system structure is called structural (algebraic) observability analysis. Note that structural observability gives a necessary condition for observability, which means that if a system is not structural observable then it is not observable. A detailed discussion on structural observability analysis of linear systems is given in Reinschke (1988). The system structure can be represented

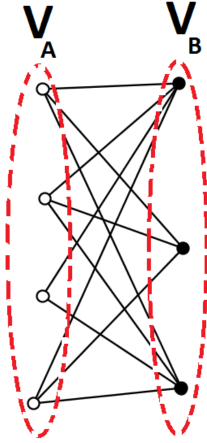

Figure 5: A bigraph. There are two sets of disjoint vertices (white and black colored). No edges among white nodes as well as black nodes.

graphically (using diagraph) and hence, graph-theoretic techniques can be used to analyze structural observability.

The system diagraph G is created in the following way. There are $n_x + n_y$ number of nodes representing state and output variables. If $< i, j >$-th element of A is not zero then there exists a directed edge from x_i to x_j. Similarly, the edges (x_i, y_j) are created by considering non-zero elements in C. The definition 1 gives the condition for output connectivity and the definition 2 defines the conditions to be satisfied for structural observability.

Definition 1 *A class of systems is said to be output-connectable*[6] *(or reachable) if in the digraph G there is a path from at least one of the output-vertices to every state-vertex. Reinschke (1988)*

Definition 2 *A class of systems is s-observable if and only if the digraph G meets the following condition:*

- *G is spanned by cacti. Lin (1974)*[7]*, Reinschke (1988)*

Violation in the definition 2 will make the system not s-observable, hence not observable. To find the minimum number of driver nodes to achieve s-observability, the minimum input theorem is used Liu et al. (2011). The first step is to create the corresponding bipartite graph of the digraph. Let $G(V, E)$ be a digraph where $V = \{v_1, v_2, \ldots, v_{n_v}\}$ and $E = \{e_1, e_2, \ldots, e_{n_e}\}$. Define two disjoint sets of nodes such that $V^+ = \{v_1^+, v_2^+, \ldots, v_{n_v}^+\}$ and $V^- = \{v_1^-, v_2^-, \ldots, v_{n_v}^-\}$. Then create a bigraph with V^+ and V^-. If $(v_i, v_j) \in E$, then (v_i^+, v_j^-) is an edge of the bigraph. A maximum (or perfect) matching in the bigraph also gives a maximum (or perfect) matching in the digraph. The minimum number of driver nodes needed to achieve the s-observability is equal to the number of unmatched v_i^+'s in the bigraph and the driver nodes are corresponding v_i's. Note that if there is a perfect matching then there is a single driver node. Matching algorithms for bipartite graphs are already implemented in NetworkX so that the minimum input theorem can be easily implemented in Python. Also refer the supplementary section to Liu et al. (2011) for more details.

Practical systems are often nonlinear, but interestingly it is possibly to apply the graph-theoretic approach discussed above for LTI systems directly to nonlinear systems Reinschke (1988), Daoutidis and Kravaris (1992), Liu et al. (2012), Boukhobza and Hemlin (2007) and Liu et al. (2011). Consider the nonlinear state space model

$$\begin{aligned} \dot{x} &= f\left(t, x, u\right), \\ y &= g\left(t, x, u\right), \end{aligned} \tag{2}$$

where $f = [f_1, f_2, \ldots, f_{n_x}]^T$, and $g = \left[g_1, g_2, \ldots, g_{n_y}\right]^T$. The digraph (G) containing $n_x + n_y$ number of nodes. If $\frac{\partial f_i}{\partial x_j} \neq 0$ then $x_i \rightarrow x_j$ exists and similarly, if $\frac{\partial g_i}{\partial x_j} \neq 0$ then $y_i \rightarrow x_j$ exists. Note that partial derivatives should be found symbolically Perera (2014), Perera et al. (2014), not numerically. The reason is that, for example even though x_j occurs in f_i, still it is possible to have numerically $\frac{\partial f_i}{\partial x_j} = 0$.

State estimation techniques — for example extended Kalman filtering — may be used to estimate unknown disturbances and unknown/uncertain parameters Simon (2006), Jazwinski (2007), Åström (2006). Let the nonlinear state space model

$$\begin{aligned} \dot{x} &= f\left(t, x, u, w, p\right), \\ y &= g\left(t, x, u, w, p\right), \end{aligned} \tag{3}$$

where $w = [w_1, w_2, \ldots, w_{n_w}]^T$ is the disturbance vector and $p = \left[p_1, p_2, \ldots, p_{n_p}\right]^T$ is the parameter vector. Assume w and p are unknown disturbances and uncertain parameters to estimated. One possibility is to write

$$\begin{aligned} \dot{w} &= 0, \\ \dot{p} &= 0, \end{aligned} \tag{4}$$

[6] Also called Y-topped Boukhobza and Hemlin (2007).

[7] Though Lin (1974) considered s-controllability, the concepts can be easily adapted to s-observability.

and then to augment w and p to the current state x. I.e. $\tilde{x} = [x, w, p]^T$. Now the augmented state space model is

$$\dot{\tilde{x}} = \tilde{f}(t, \tilde{x}, u), \quad (5)$$
$$y = \tilde{g}(t, \tilde{x}, u).$$

$\tilde{f}$ and $\tilde{g}$ are then used in s-observability analysis as already explained using graph-theoretic techniques.

4. Python implementation

4.1. Modelica, JModelica.org and CasADi options

Modelica is becoming a standard tool for modeling large-scale complex physical systems. CasADi is a symbolic framework — a CAS tool — for numerical optimization and it is available to use it within Python. Modelica models — which result in differential algebraic equations, DAEs — can be imported to Python via CasADi and make symbolic DAEs available for general use in Python. See Perera et al. (2014) and Perera (2014). CasADi comes with JModelica.org and it may be the easiest way of accessing CasADi in Python.
JModelica.org provides three Python packages: `pymodelica`, `pyfmi` and `pyjmi`. `pymodelica` is for compiling (or model export) Modelica models while other two packages are for model import. `pyfmi` is for creating model objects according to FMI (Functional Mock-Up Interface) standards which is not at our interest here in this paper. `pyjmi` is for JModelica.org platform specific model importing. The relevant choices for exporting and importing are: the compiler `compile_fmux` (from `pymodelica`) for compiling and `CasadiModel` (from `pyjmi`) for importing.[8]

4.2. Structure of the Python script

The skeleton of the Python script is depicted in figure 6. First, the system model is encoded as a Modelica model. The Modelica model is then compiled and imported back to Python as a `CasadiModel` model object. The imported model is a symbolic flat representation of the Modelica model. Now using the CasADi Python package, necessary symbolic Jacobian matrices — which appeared in section 3 — are found. Once the Jacobian matrices are available, corresponding digraphs can be easily generated by means of NetworkX and PyGraphViz Python packages.[9] NetworkX supports to create four types of graph objects: Graph, DiGraph, MultiGraph, and MultiDiGraph.[10] In NetworkX, Graph and DiGraph graph objects are used only for graphs without multiple edges. To create graphs with multiple edges MultiGraph/MultiDiGraph graph objects should be used. It is clear that for s-observability analysis MultiDiGraph graph objects must be used. NetworkX provides many network algorithms related to: matching, bipartite graph related, strongly connectivity, cycles, tree, etc. The PyGraphviz Python package can be used as the layout tool.[11] The Matplotpib Python package may also be used for network drawing. The NetworkX and PyGraphviz network objects are convertible to each other.

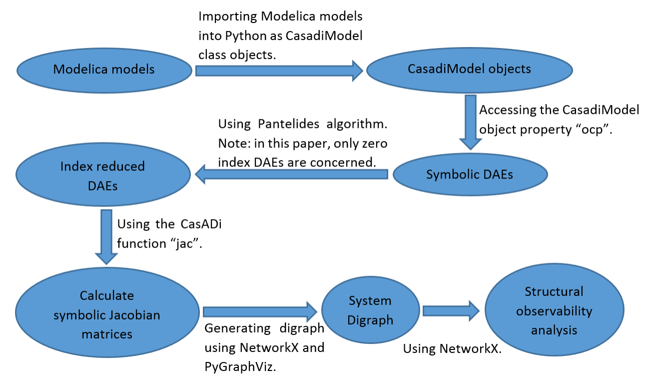

Figure 6: Structure of the Python script.

Let the Modelica model be stored in the file “MyModel.mo” and the model name is “mymodel”. The compilation is done using the Python code given below:[12]

```
# Import compiler compile_fmux
from pymodelica import compile_fmux
# Compile Modelica model
file_name = 'MyModel.mo'
model_name = 'mymodel'
compile_fmux(model_name,file_name)
```

The compiled model object is a “.fmux” file with the name “mymodel” and the model object is imported as a `CasadiModel` object. The code is given below:

```
from pyjmi import CasadiModel
casadiModelObject = \
    CasadiModel('mymodel.fmux')
```

[8] Often index of DAEs is greater than one (i.e. higher index problems.). In such cases, index should be reduced zero using Pantelides algorithm before applying the concept discussed in this paper. See Pantelides (1988) and Cellier and Kofman (2006).

[9] See in http://networkx.github.io/ and http://pygraphviz.github.io/.

[10] “Multi” indicates that the graph object support multiple edges.

[11] See in http://pygraphviz.github.io/.

[12] The complete Python code will be available on request.

```
# Get flat ocp representation
ocp = casadiModelObject.ocp
```

Also, in order to use the CasADi package, it is imported as given below:

```
from casadi import *
from casadi.tools import *
```

`ocp` contains the information about the flattened symbolic model. For example `ocp.x`, `ocp.z`, `ocp.pi`, `ocp.pd`, `ocp.pf`, `ocp.t`, `ocp.u`, `ocp.ode`, and `ocp.alg` give respectively dynamic vector, algebraic state, independent parameter vector, dependent parameter vector, free parameter vector, time, input vector, vector of ODEs and vector of algebraic equations. Also `casadiModelObject.dx` gives dynamic state derivative vector. The main ingredient for generating a MultiDiGraph object is to have functions f and g which are given in equation 3 or $\tilde{f}$ and $\tilde{g}$ in equation 5. f is defined as an `SXFunction` class instance, say `ffun`. See below for the Python code:

```
# Define ODEs
f = ocp.ode
# Create an SXFunction for f
ffun = SXFunction([t,vertcat(xDot),vertcat(x),\
    vertcat(u)],[f])
ffun.init()
```

Now, consider how to create a `MultiDiGraph`. The Jacobian matrix $\frac{\partial f}{\partial x}$, which is given by `ffun.jac(1)`, contains the information about the dependencies among state variables. In a similar way, g is defined as an `SXFunction`. Then $\frac{\partial g}{\partial x}$ gives information to construct the dependencies among state and output variables. the NetworkX and PyGraphViz packages are imported in the following way:

```
import networkx as nx
import pygraphviz as pgv
```

`G = nx.MultiDiGraph()` creates the MultiDiaGraph object with no edges and nodes. In order to add nodes for state, input and output variables use the following code:

```
G = nx.MultiDiGraph()
# Create state vertices
for i in x:
    G.add_node('{0}'.format(x[i]))
# Create input vertices
for j in u:
    G.add_node('{0}'.format(u[j]))
# Create output vertices
for k in y:
    G.add_node('{0}'.format(y[k]))
```

In order to add edges we can use the following code:

```
# Create edges among states
for i in range(n_x):
    for j in range(n_x):
        if isEqual(A[i,j],SX('0')) == False:
            G.add_edge('{0}'.format(x[j]),\
    ''.format(x[i])
# Create edges among states and inputs
for i in range(n_x):
    for j in range(n_u):
        if isEqual(B[i,j],SX('0')) == False:
            G.add_edge('{0}'.format(x[j]),\
    '{0}'.format(u[i])
            # Create edges among states \
    and inputs
for i in range(n_y):
    for j in range(n_x):
        if isEqual(C[i,j],SX('0')) == False:
            G.add_edge('{0}'.format(y[j]),\
    '{0}'.format(x[i])
```

Additionally, it is useful to do some formatting on nodes/edges. For example, states, input and output nodes are in different colors and shapes. NetworkX graph object can be converted to PyGraphViz AGraph objects using `Gp = nx.to_agraph(G)`. See the code below:

```
Gp = nx.to_agraph(G)
Gp.write("file.dot")
Gp.layout()
Gp.layout(prog='dot')
G.draw('file.png')
```

In order to have a better structured code, several new functions may be defined within the `CasadiModel` class: symbolicLinearization(), symbolicDAEs(), indexReduction(), createNodes(), createEdges(), generateGraph(), decomposeGraph(), Y_Topped(), Max_Matching(), etc. Now these functions can be called as for instance `casadiModelObject.createNodes()`. symbolicDAEs() creates symbolic functions for ODEs and algebraic equations. indexReduction() is used for index reduction. Symbolic Jacobian matrices are found by symbolicLinearization(). Based on Jacobian matrices the nodes and the edges of the digraph are generated using createNodes() and createEdges(). generateGraph() creates a NetworkX and a PyGraphViz graph objects as well as it creates a 'dot'[13] file. decomposeGraph() decomposes the digraph into strongly connected components. To check the conditions given in the definition 2 Y_Topped() and Max_Matching() are used.
Let us define some terms (based on Anh (2012), Liu et al. (2012), and Liu et al. (2011)). State nodes which

[13] See in http://www.graphviz.org/.

are directly connected to output nodes are called controlled nodes. A controlled node with just a single connection to an output is called a driver node.
As a summary to this section, the following points are made: (1) a Modelica model is created, (2) import the dynamic model as a CasadiModel object model and use `casadi` to find symbolic Jacobian matrices of symbolic DAEs (after reducing the index if needed), (3) generate a digraph using networkx and pygrapviz, (4) use graph theories to analyze the digraph.

5. Industrial Application Case

The Copper electro-winning process at Glencore Nikkelverk, Kristiansand, Norway is considered. The process consists of four sections: (i) the slurrification where the calcine containing mostly copper oxide is slurrified using recycled anolyte flow, which containing sulfuric acid, taken from the electrowinning section, (ii) the leaching section where sulfuric acid is added to the slurry in order to leach more copper into the solution, (iii) the purification section where the slurry is first filtered to extract the solution, which contains copper sulphate ($CuSO_4$), followed by the cementation and fine filtering processes, and (iv) the electrowinning section where the solution containing Cu^{2+} is electrolyzed to release solid copper at the cathode. For a detailed discussion and a mechanistic model for the Copper electro-winning process given in Lie and Hauge (2008). Figure 7 in appendix C gives the flow sheet [14]. The system model is in the form of equation 3 while the augmented model — by taking $\dot{p} = 0$ and w as slowly varying (i.e. $\dot{w} \approx 0$) — is in the form of equation 5. The nodes corresponding to the parameters and disturbances have directed edges always directing towards them starting from either output/state nodes. I.e. possible edges ending at parameter/disturbance nodes are $x_i \to p_j$, $y_i \to p_j$, $x_i \to w_j$ and $y_i \to w_j$. In the following discussion, it is shown that how to implement the procedure given in figure 6 in relation to the dynamic model given in Lie and Hauge (2008). The tank-volume (or level) dynamics are neglected in the original model. But, the tank-volume dynamics of the electro-winning section is included and the new model is considered in this paper. See equations 6, 7 and 8. It is assumed that the liquid level of the electro-winning tank is a constant. Except $\dot{V}_{ed2w}$, $\dot{V}_{ew2m}$ and $\dot{V}_{vap}$, the rest of the volumetric flow rates are known.

$$\dot{V}_{ed} = \dot{V}_{em2d} + \dot{V}_{p2e} - \dot{V}_{ed2w} - \dot{V}_{ed2m} \tag{6}$$

[14] Taken from Lie and Hauge (2008).

$$\dot{V}_{em} = \dot{V}_{ed2m} + \dot{V}_{ew2m} + \dot{V}_{w2em} - \dot{V}_{e2s} - \dot{V}_{em2bl} - \dot{V}_{em2d} \tag{7}$$

$$\dot{V}_{ew} = \dot{V}_{ed2w} - \dot{V}_{ew2m} - \dot{V}_{vap} = 0 \tag{8}$$

Equations 9, 10, 11, and 12 are resulted by applying (static) mass balances to the slurrification, leaching and purification sections.

$$\dot{V}_{s2l} = \dot{V}_{e2s} \tag{9}$$

$$\dot{V}_{l,o}^{(1)} = \dot{V}_{e2s} + \dot{V}_{a} \tag{10}$$

$$\dot{V}_{w2l} = \dot{V}_{p2e} - \dot{V}_{e2s} - \dot{V}_{a} \tag{11}$$

$$\dot{V}_{l2p} = \dot{V}_{p2e} \tag{12}$$

There are 41 states, 3 inputs, 3 outputs, 8 disturbances (some of the disturbances are measured.) and 5 parameters in the model. See below:

- states in the slurrification section (2 tanks in series):
  $\rho_{s,i}^{j};\ i \in \{CuO, CuSO_4, H_2SO_4\}$,
  $j \in \{(1),(2)\}$
- states in the leaching section (5 tanks in series):
  $\rho_{l,i}^{j};\ i \in \{CuO, CuSO_4, H_2SO_4\}$,
  $j \in \{(1),(2),(3),(4),(5)\}$
- states in the purification section (6 tanks in series):
  $\rho_{pb,i}^{j};\ i \in \{CuSO_4, H_2SO_4\}$,
  $j \in \{(1),(2),(3),(4),(5),(6)\}$
- states in the electro-winning section:
  - the dilution tank:
    $\rho_{ed,i};\ i \in \{CuSO_4, H_2SO_4\}$
    V_{ed}
  - the electro-winning tank:
    $\rho_{ew,i};\ i \in \{CuSO_4, H_2SO_4\}$
  - the mixing tank: $\rho_{em,i}$;
    $i \in \{CuSO_4, H_2SO_4\}$
    V_{em}
- inputs: $\dot{m}_c$, $\dot{V}_{e2s}$, and $\dot{V}_a$.
- outputs:
  $\rho_{pb,H_2SO_4}^{(3)}$,
  $\frac{M_{Cu}}{M_{CuSO_4}} \cdot \rho_{ew,CuSO_4}$,
  and $\rho_{ew,H_2SO_4} + \frac{M_{H_2SO_4}}{M_{CuSO_4}} \cdot \rho_{ew,CuSO_4}$.
- measured disturbances: $\dot{V}_{ed2m}$, $\dot{V}_{em2d}$, $\dot{V}_{em2bl}$, $\dot{V}_{p2e}$, $\dot{V}_{w2em}$ and I

- unmeasured disturbances: $\dot{V}_{w2l}$, $\dot{V}_{s2l}$, $\dot{V}_{l,o}^{(1)}$, $\dot{V}_{l2p}$, $\dot{V}_{ew2m}$, $\dot{V}_{ed2w}$, $\dot{V}_{vap}$, $x_{c,\mathrm{Cu}}$ and $\rho_{a,\mathrm{H_2SO_4}}$.
- parameters (all are unknown): k, η, $\epsilon_{ps}^{(1)}$, $\epsilon_{ps}^{(2)}$, and $\epsilon_{ps}^{(3)}$.

Now a Modelica model is created for the (updated) dynamic model explained above and also augment unknown independent parameters as states: $\frac{d}{dt}k = 0$, $\frac{d}{dt}\eta = 0$, $\frac{d}{dt}\epsilon_{ps}^{(1)} = 0$, $\frac{d}{dt}\epsilon_{ps}^{(2)} = 0$ and $\frac{d}{dt}\epsilon_{ps}^{(3)} = 0$. Note that, it may not be possible to augment all the unmeasured disturbances, but a subset of them. The reason is possible algebraic relationships among unmeasured disturbances, states, inputs and measured disturbances.[15] $\dot{V}_{w2l}$, $\dot{V}_{s2l}$, $\dot{V}_{l,o}^{(1)}$ and $\dot{V}_{l2p}$ can be expressed in terms of inputs and measured disturbances while $x_{c,\mathrm{Cu}}$ and $\rho_{a,\mathrm{H_2SO_4}}$ cannot, therefore $x_{c,\mathrm{Cu}}$ and $\rho_{a,\mathrm{H_2SO_4}}$ should be augmented. $\dot{V}_{ew2m}$, $\dot{V}_{ed2w}$ and $\dot{V}_{vap}$ are dependent each other, therefore 2 out of 3 should be augmented. Let us choose to augment $\dot{V}_{ew2m}$ and $\dot{V}_{ed2w}$, thereby $\dot{V}_{vap}$ becomes a function of the augmented states. See below:

- a subset of unmeasured disturbances are augmented as new state variables: $\frac{d}{dt}\dot{V}_{ed2w} = 0$, $\frac{d}{dt}\dot{V}_{ew2m} = 0$, $\frac{d}{dt}x_{c,\mathrm{Cu}} = 0$ and $\frac{d}{dt}\rho_{a,\mathrm{H_2SO_4}} = 0$. Also, $\dot{V}_{vap} = \dot{V}_{ed2w} - \dot{V}_{ew2m}$.

Now, including augmented states, there are 48 (39 states in the original model, 5 unknown parameters and 4 unmeasured disturbances) states. The structure of the Modelica code is given in appendix A.
The structure of the Python script which is used to generate the digraph for the structural observability analysis is given in appendix A. The Python script creates the digraph G, which is given in figure 8 in appendix C. G can be decomposed into SCCs using the function `decomposeGraph()`. See figure 9 in appendix C. There are two SCCs with more than one node. Each SCC is colored with different colors. It is possible to check whether G is output connected using `Y_Topped()` (see definition 1). Here we use `networkx.all_simple_paths()` and this function gives all possible paths starting from a given node and ending at a given node if any. See the script given below for the definition of `Y_Topped()`:

```
def Y_Topped(self):
        Gnx = self.Gnx
        x = self.x
        k = 'Y-topped'
        for j in x:
            index = 0
            dummy_array = N.zeros(5)
            for i in ['y1','y2','y3','y4','y5']:
                if list(ntwx.all_simple_paths(\
Gnx,source=i,target=str(j)))==[]:
                    dummy_array[index] = 0
                else:
                    dummy_array[index] = 1
                index = index + 1
            if N.max(dummy_array) == 0:
                k = 'Not Y-topped'
                break
            else:
                pass
        print k
```

Consider figure 9. The two elementary paths $y_1 \to V_{ed} \to \dot{V}_{ed2w}$ and $y_2 \to V_{em} \to \dot{V}_{ew2m}$ are two stems. Since neither V_{ed} nor V_{em} has edges going out from them, both are cacti without any buds. The SCC with plain-green colored nodes has no incoming edges from the other SCC which is in red, hence there must be at least one measurement node which is connected to a node in the plain-green colored SCC and y_4 and y_5 satisfy this condition.[16] Since y_4 has only one edge, y_4 should be used to create the cacti, $y_4 \to \rho_{ed,\mathrm{CuSO_4}} \to \rho_{pb,\mathrm{CuSO_4}}^{(3)} \to \rho_{pb,\mathrm{CuSO_4}}^{(2)} \to \rho_{ps,\mathrm{CuSO_4}}^{(3)} \cdots \to \rho_{s,\mathrm{CuSO_4}}^{(2)} \to \rho_{ps,\mathrm{CuSO_4}}^{(1)} \to \rho_{em,\mathrm{CuSO_4}}$. The remaining two measurement nodes y_3 and y_5 are not yet used. Since there are many nodes with only incoming edges — yellow color edges η, ϵ_1, $\cdots$, $x_{c\mathrm{Cu}}$ —, it is impossible to find cacti starting from y_3 and y_5. Therefore, no spanning cacti is found for G. Thereby, the augmented system model is not structurally observable and hence not observable. In other words, it is impossible to completely estimate the augmented system state using given measurements.

6. Conclusion

We have demonstrated how to implement structural observability analysis, in the view of graph-theoretic approach, for large scale complex dynamic system in Python by using NetworkX, PyGraphViz Python packages as well as CasADi's Python front-end. The main result is how to find the spanning cacti for a given digraph in order to find the minimum number of driver nodes. Modelica is used for modeling and it is a standard tool for modeling large scale complex dynamic system. CasADi supports to import Modelica models into Python as flattened symbolic DAEs making it possible to use Modelica models in general use. Importantly, all the software tools which are used in our development are free.

[15] The state augmentation should be done after index reduction if the problem is high index. The problem under consideration of this paper is a zero index problem, hence index reduction is not necessary.

[16] The SCC with plain-green colored nodes is the root-SCC. Liu et al. (2012)

References

Anh, N. T. T. *Spanning Cacti for Structurally Controllable Networks*. Master's thesis, Department of Mathematics, National University of Singapore, 2012.

Bondy, A. and Murty, U. S. R. *Graph theory*. Graduate texts in mathematics. Springer, 2008.

Boukhobza, T. and Hemlin, F. Observability analysis for structured bilinear systems: a graph-theoretic approach. *Automatica*, 2007. 43(11). doi:10.1016/j.automatica.2007.03.010.

Cellier, F. E. and Kofman, E. *Continuous System Simulation*. Springer, 2006.

Daoutidis, P. and Kravaris, C. Structural evaluation of control configurations for multivariable nonlinear processes. *Chemical Engineering Science*, 1992. 47:1091–1107. doi:10.1016/0009-2509(92)80234-4.

Jazwinski, A. H. *Stochastic Processes and Filtering Theory*. Dove Publications, Inc., Mineola, New York, 2007.

Lie, B. and Hauge, T. A. Modeling of an industrial copper leaching and electrowinning process, with validation against experimental data. *Proceedings SIMS 2008, 49th Scandinavian Conference on Simulation and Modeling*, 2008.

Lin, C. T. Structural controllability. *IEEE Transactions on Automatic Control*, 1974. 19(3). doi:10.1109/TAC.1974.1100557.

Liu, Y.-Y., Slotine, J.-J., and Barabàsi, A.-L. Controllability of complex networks. *Nature*, 2011. 473:167–173. doi:doi:10.1038/nature10011.

Liu, Y.-Y., Slotine, J.-J., and Barabàsi, A.-L. Observability of complex systems. *Proceedings of the National Academy of Sciences of the United States of America*, 2012. 110(7):2460–2465. doi:10.1073/pnas.1215508110.

Pantelides, C. C. The consistent initialization of differential-algebraic systems. *SIAM Journal on Scientific Computing*, 1988. 9(2). doi:10.1137/0909014.

Perera, A. Using casadi for optimization and symbolic linearization/extraction of causality graphs of modelica models via jmodelica.org. Technical Report HiT_rapport_5, Telemark University College, Kjølnes ring 56, P.O. Box 203, N-3901 Porsgrunn, Norway., 2014. URL https://teora.hit.no/handle/2282/2175.

Perera, A., Pfeiffer, C., Hauge, T. A., and Lie, B. Making modelica models available for analysis in python control systems library. *Proceedings SIMS 2014, 55th Scandinavian Conference on Simulation and Modeling*, 2014.

Åström, K. J. *Introduction to Stochastic Control Theory*. Dove Publications, Inc., Mineola, New York, 2006.

Reinschke, K. J. *Multivariable control: a graph theoretic approach*. Lecture notes in control and information sciences. Springer-Verlag, Berlin, New York, 1988. URL http://opac.inria.fr/record=b1086834.

Simon, D. *Optimal State Estimation: Kalman, H Infinity, and Nonlinear Approaches*. John Wiley & Sons, Inc., Hoboken, New Jersey, 2006.

Appendices

A. Structure of the Modelica model

```
model CopperPlant
// Augment unknown parameters as states
Real k;
Real eta;
Real eps_1;
Real eps_2;
Real eps_3;
// Augment disturbances as states
Real Vd_ed2w;
Real Vd_ew2m;
Real x_cCu;
Real rho_aH2SO4;
// Define sates
Real V_ed;
Real V_em;
Real rho_s1CuO;
Real rho_s1CuSO4;
Real rho_s1H2SO4;
Real rho_s2CuO;
Real rho_s2CuSO4;
...
...
Real rho_edCuSO4;
Real rho_edH2SO4;
Real rho_ewCuSO4;
Real rho_ewH2SO4;
Real rho_emCuSO4;
Real rho_emH2SO4;
// Define known parameters
```

```
parameter Real z_Cu = 2;
parameter Real C = 96485.0;
parameter Real V_s1 = 7400;
...
...
// Define inputs/measured disturbances
input Real md_c;
input Real Vd_e2s;
input Real Vd_a;
input Real Vd_p2e;
input Real Vd_ed2m;
input Real Vd_em2d;
input Real Vd_em2bl;
input Real Vd_w2em;
input Real I;
// Define dependent disturbances
// in terms of inputs/states/parameters
Real Vd_w2l = Vd_p2e - Vd_e2s - Vd_a;
Real Vd_vap = Vd_ed2w - Vd_ew2m;
Real Vd_s2l = Vd_e2s;
Real Vd_l1o = Vd_e2s + Vd_a;
Real Vd_l2p = Vd_p2e;
// Define other variables as needed
...
...
// Define equations
equation
der(V_ed) = Vd_em2d + Vd_p2e - Vd_ed2w - Vd_ed2m;
...
...
der(k) = 0;
der(eta) = 0;
der(eps_1) = 0;
der(eps_2) = 0;
der(eps_3) = 0;
der(x_cCu) = 0.0;
der(rho_aH2SO4) = 0.0;
der(Vd_ed2w) = 0;
der(Vd_ew2m) = 0;
end CopperPlant;
```

B. Structure of the Python code

```
################################################
## JModelica.org version 1.12 is used        ##
## Several functions are added to CasadiModel ##
## class such as: symbolicDAEs(),            ##
## symbolicLinearization(), createNodes(),   ##
## ..., and Max_Matching_BP().               ##
## Pygraphviz, Networkx, Pydot, Pyparsing,   ##
## and Casadi Python packages are used.      ##
################################################
# Import the compiler
from pymodelica import compile_fmux
# Compiling Modelica model
fmux = compile_fmux(\
    'CopperPlantPackage.CopperPlant',\
    'CopperPlant.mo')
# Importing compiled model
from pyjmi import CasadiModel
model = CasadiModel(fmux)
# Creating symbolic DAEs
model.symbolicDAEs()
# Symbolic linearization
model.symbolicLinearization()
# Creating nodes
model.createNodes()
#Creating edges
model.createEdges()
# Creating digraph
model.generateGraph()
# Decomposing digraph into SCCs
model.decomposeGraph()
# Checking output connectivity
model.Y_Topped()
# Finding maximum matching
model.Max_Matching_BP()
```

C. Flow sheet and digraphs

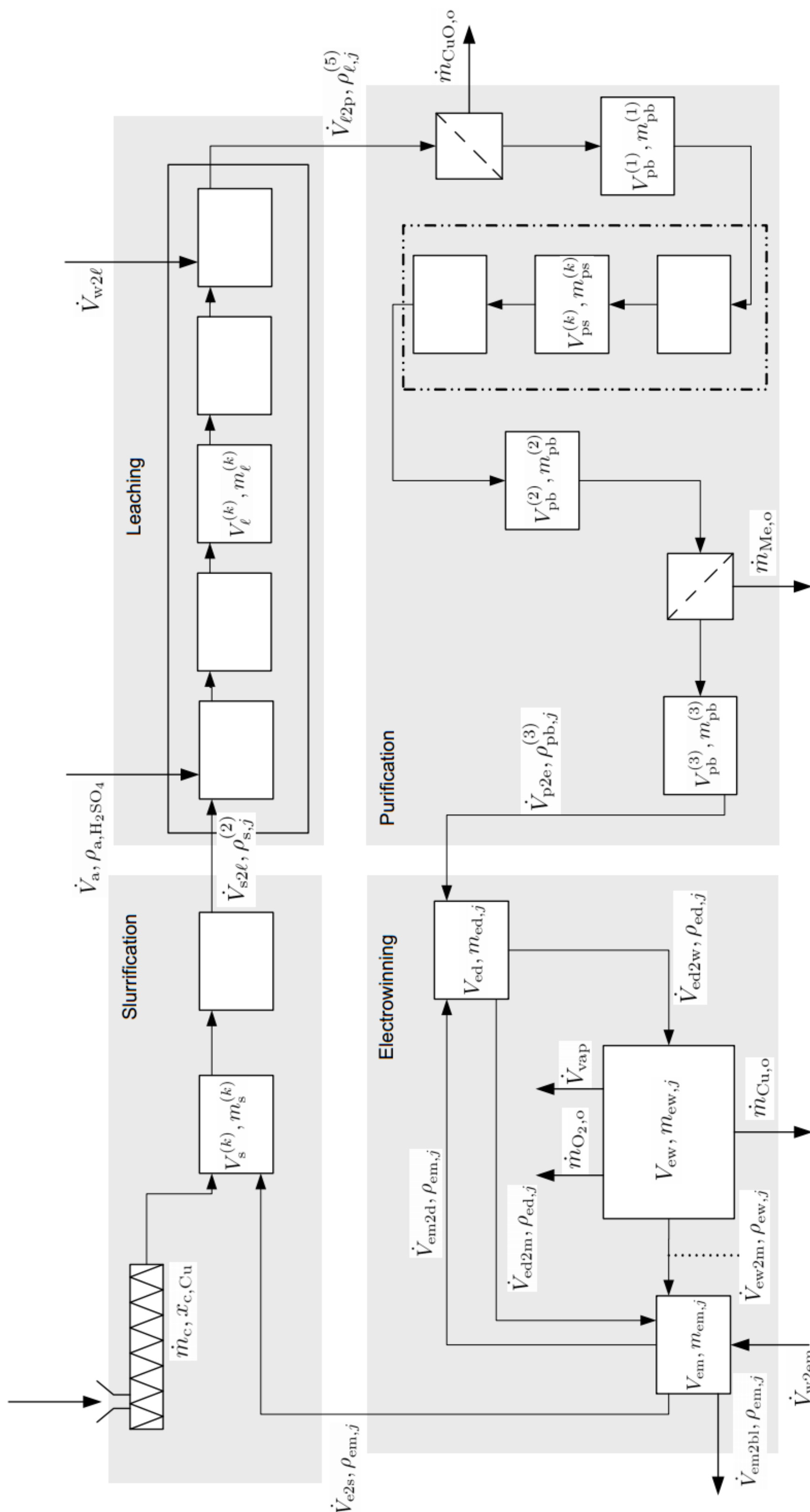

Figure 7: The process flow sheet for the Copper electro-winning process.

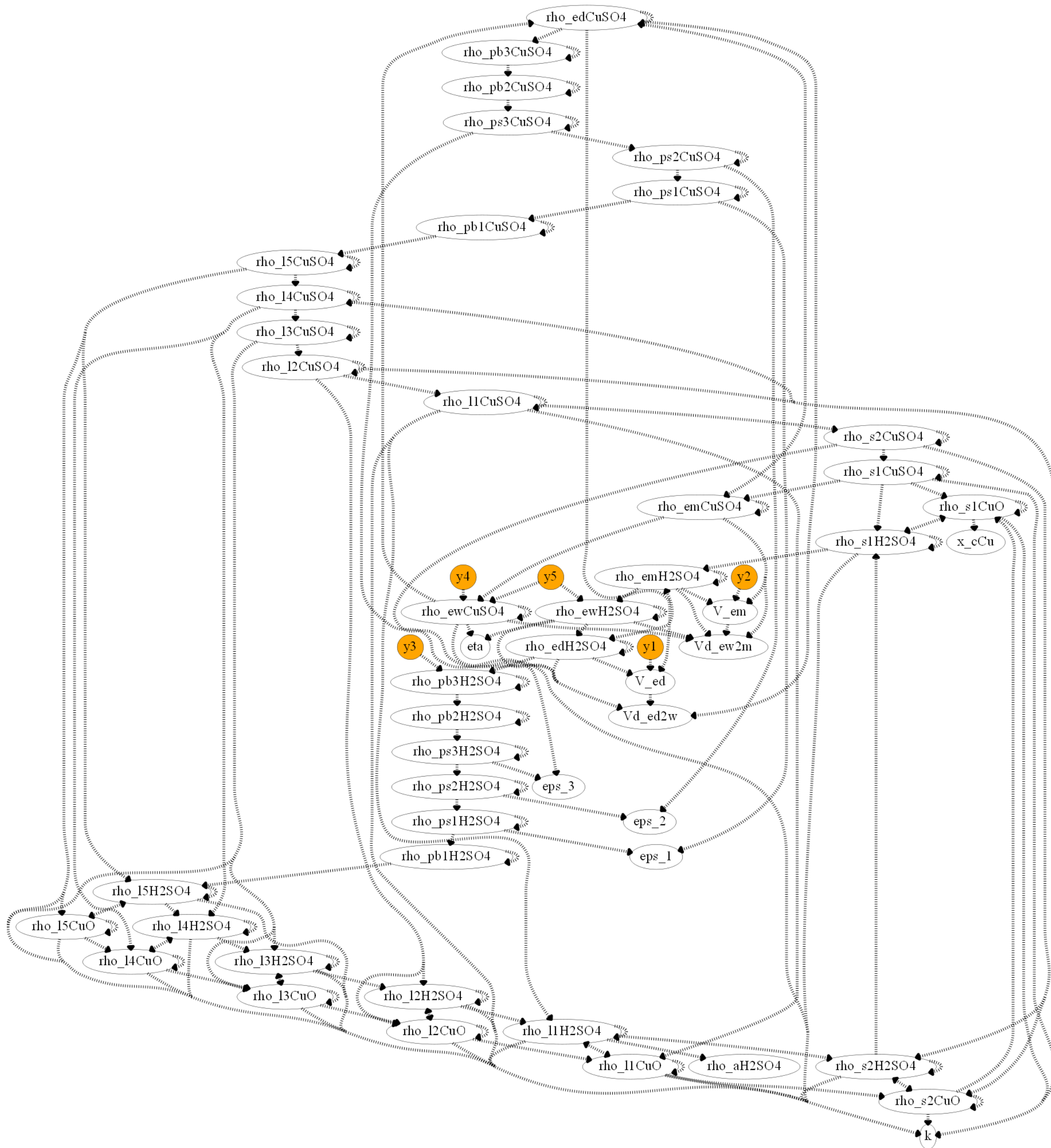

Figure 8: The digraph (G) for the Copper plant model.

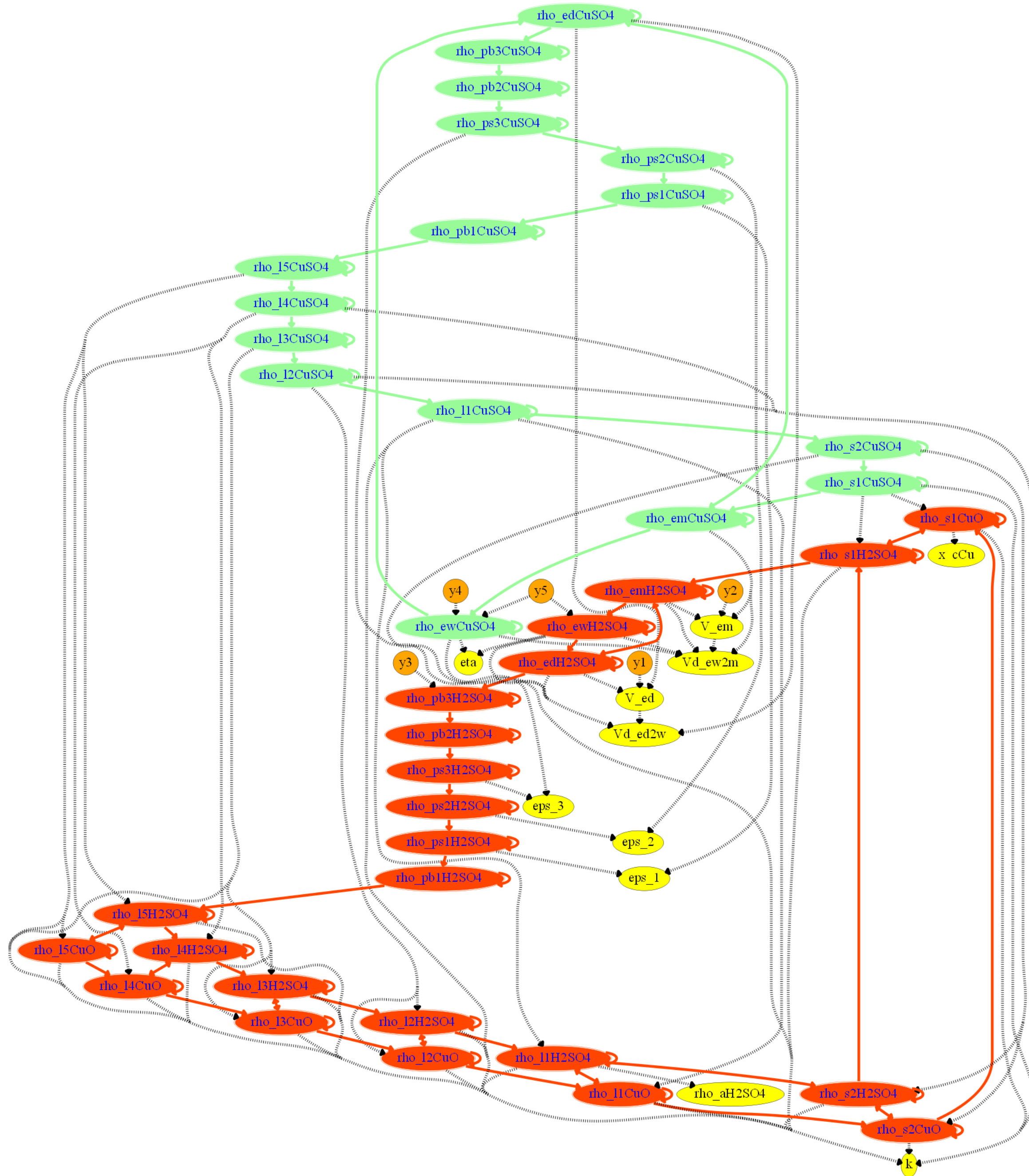

Figure 9: Strongly connected components (SCCs) of G.