Master's thesis

NTNU
Norwegian University of Science and Technology
Faculty of Natural Sciences
Department of Chemical Engineering

Andreas Johannesen

Initialization Methods for large Process Models

Master's thesis in Chemical Engineering
Supervisor: Heinz A Preisig, Arne Tobias Elve
June 2019

NTNU
Norwegian University of
Science and Technology

Andreas Johannesen

Initialization Methods for large Process Models

Master's thesis in Chemical Engineering
Supervisor: Heinz A Preisig, Arne Tobias Elve
June 2019

Norwegian University of Science and Technology
Faculty of Natural Sciences
Department of Chemical Engineering

Summary

Model building is an essential part of any model-driven research. Many aspects of modeling are time-consuming, one of which being the initialization when the models become large. This thesis presents the implementation of an instantiating class for adequately and efficiently formulating and initializing process models. To develop the models, the class combines two model components, namely a directed graph, providing the model structure, and a mathematical framework, providing the possible variables and equations the model can contain. The directed graph consists of nodes and arcs, where the nodes represent a capacity containing conserved quantities such as mass and energy, while the arcs represent the transport of these quantities between the nodes. The nodes have the ability to be grouped into a group node by assuming equal properties and thereby also create a group of arcs connecting the children of the group node. To build the mathematical model, a set of states has to be chosen. The states are variables containing enough information to describe the entire system. By using the states as a starting point, the class formulates a mathematical model only dependent on constants and the states, giving an initial value problem with zero degrees of freedom. The acquired constants and initial values of the states need to be assigned values. Various initialization schemes can be executed initializing on either single nodes and arcs or groups, giving the same value to all entities contained in that group. The model with complete initialization is exported to a modeling software, providing the necessary information to simulate the model. The initialization is also saved as a case in a repository, and the class offers the ability to load the case into a model which is using the same graph and mathematical framework. The initial states of the model often have non-trivial values making it difficult to assign the correct values. In many cases, equations already present in the mathematical model is used to calculate the states by the use of more accessible variables. The thesis also looks into the possibility of locating these equations and check if they can be solved by a proposed variable set. A model example with the use of the class is presented to show proof of concept.

Sammendrag

Modellbygging er en viktig del av enhver modelldrevet forskning. Mange aspekter ved modellering er tidkrevende, hvorav en er initialiseringen når modellene blir store. Denne oppgaven presenterer implementeringen av en instanserende klasse for tilstrekkelig og effektiv formulering og initialisering av prosessmodeller. For å formulere modellene, kombinerer klassen to modellkomponenter, nemlig en rettet graf, som gir modellstrukturen og et matematisk rammeverk, som gir mulige variabler og ligninger som modellen kan inneholde. Den rettede grafen består av noder og buer, hvor nodene representerer en kapasitet som inneholder konserverte mengder som masse og energi, mens buene representerer transporten av disse mengdene mellom nodene. Nodene har muligheten til å bli gruppert i en gruppenode ved å anta like egenskaper, og dermed også lage en gruppe buer som forbinder barnenodene til gruppenoden. For å bygge den matematiske modellen må et sett tilstander velges. Tilstandene er variabler som inneholder nok informasjon til å beskrive hele systemet. Ved å bruke tilstandene som utgangspunkt, formulerer klassen en matematisk modell som bare er avhengig av konstanter og tilstandene, og gir et initialverdiproblem med null frihetsgrader. Konstanter og initielle verdier av tilstandene må tilordnes verdier. Forskjellige initialiseringsordninger kan utføres på enten enkeltnoder og enkeltbuer eller grupper, og gir samme verdi til alle enheter som finnes i den gruppen. Modellen med fullstendig initialisering blir eksportert til en modelleringsprogramvare, og gir den nødvendige informasjonen for å simulere modellen. Initialiseringen lagres også som et tilfelle i et lager, og klassen gir muligheten til å laste saken til en modell som bruker samme graf og matematiske rammeverk. De opprinnelige tilstandene til modellen har ofte ikke-trivielle verdier som gjør det vanskelig å tilordne de riktige verdiene. I mange tilfeller brukes ligninger som allerede er tilstede i den matematiske modellen til å beregne tilstandene ved bruk av mer tilgjengelige variabler. Avhandlingen ser også på muligheten for å finne disse ligningene og se om de kan løses av et foreslått variabelt sett. Et modelleksempel med bruk av klassen presenteres for å vise bevis av konsept.

Preface

This thesis has been written as a part of the degree of Master Of Science at Norwegian University of Science and Technology. The work has been carried out at the Department of Chemical Engineering with Professor Heinz A Preisig as supervisor and Arne Tobias Elve as co-supervisor from January 2019 to June 2019.

I want to express my sincerest gratitude towards Heinz and Tobias for the guidance and help provided on this thesis. Working with this project has been an invaluable experience. Thanks to my family for the support and encouragement you have given through this journey. Finally, I would like to thank my friends who have made these years some of the most memorable years of my life. Thank you.

Declaration of compliance

I hereby declare this thesis as an independent work in agreement with the exam rules, and regulations of the Norwegian University of Science and Technology.

Trondheim June 16, 2019

Andreas Johannesen

Table of Contents

List of Tables

List of Figures

Abbreviations

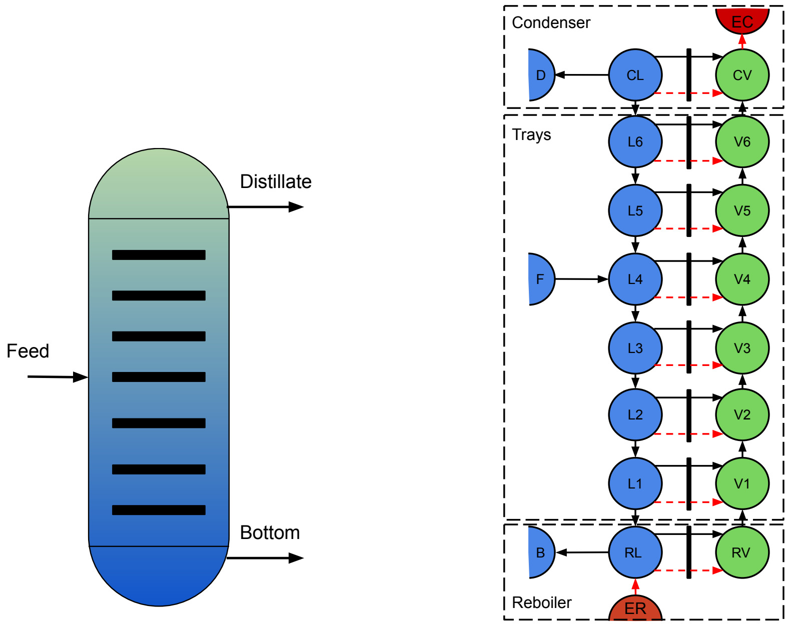
GUI	=	Graphical user interface
PDE	=	Partial differential equation
ODE	=	Ordinary differential equation

Chapter 1

Introduction

1.1 Background and motivation

Modeling, as a skill, has become an important tool in a process engineers repertoire. Model-based research, in comparison to experimental-based research, is often less time consuming and less expensive. Models also allow us to explore areas well beyond the possibilities of experiments, making it generally more practical than experimental work. Even though modeling usually is the more effective option, both the process of building the model and simulating can be quite time-consuming. The modeler has to formulate the model, implement it in a programming language, and initialize it. To make these tasks easier, different software has been developed, such as *MODEL.LA* (Stephanopoulos et al., 1990; Bieszczad, 2000) and *ASCEND* (Piela et al., 1991). The *ProcessModellerSuite* (Elve and Preisig, 2019), which this thesis is a part of, is also such a software. The *ProcessModellerSuite* uses a context-dependent ontology that structures a mathematical framework for which the process models are modeled in (Elve, 2015), an idea that was first applied in *MODKIT* (Bogusch et al., 2001; Yang and Marquardt, 2004) that utilized *OntoCAPE* (Marquardt et al., 2010) in the implementation. The modeler interactively generates a graph-based model providing information on the communication between the principal components of the model. An example of such a graph-based model is shown in (1.1b), and is termed the model topology. The model is of a distillation column with the physical outline shown in (1.1a). The topology breaks the physical model down into smaller parts. Breaking down a distillation column, creates three main sections, namely the condenser, the reboiler, and the tray section. Each of the sections will have a liquid and a gas phase, needing to be separated in the topology. Each of the sections will have a source or drain where mass is transported from or to, and the condenser and reboiler needs an energy drain and source, respectively. The tray section can also be divided into smaller sections for each tray. The resulting topology then becomes as depicted in Figure (1.1b).



(a) **(b)**

Figure 1.1: The physical outline of a distillation column with its matching topology. A liquid with 2 or more components is transported from the feed (F) into the column on tray 4. L1-L6 represent the liquid part of each tray in the column while V1-V6 represent the vapor part. The reboiler is located on the bottom with the drain for the bottom product to B. The liquid in RL is heated by the energy source (ER). The difference in volatility will make one of the components in the mixture vaporize at a lower temperature than the rest, resulting in vapor with higher a fraction of that component transported to the condenser at the top. The condenser cools the vapor and transports some of the liquid to the distillate outlet, D, while the rest cycle back down.

The topology consists of two principal components, namely nodes and arcs, the circles and arrows, respectively. The nodes represent control volumes containing conserved quantities, while the arcs represent the transport between the nodes. The software combines the information from the graphs and the ontology to generate executable program code for simulation of the model. Generation of executable code for a modeler to use has also been the focus of more recent software, such as *Mobatec* (Westerweele and Laurens, 2008), *Modeller* (Westerweele, 2003) and *MOSAIC* (Kuntsche et al., 2011). To simulate the model, initial conditions and various properties for each node and arc, based on the ontology, has to be set by the modeler. For smaller systems, this is not a very large task, but as models get larger and the complexity increases, this becomes quite a cumbersome task. Therefore, the main objective of this thesis is to create an automated instantiating procedure for the mathematical representation of the model to provide an easier and faster initialization process.

1.2 Goals

The main concept used to improve initialization time is already a fundamental part of the *ProcessModellerSuite*, something that separates it from the previously mentioned software, and the key component that makes this modeling approach different from using software based on unit operations. Unit operations-based software uses a predefined library of blocks where every block represents an input/output model of a unit operation. The blocks generated by the *ProcessModellerSuite* are fundamental entities, meaning that the model can operate as a contained entity without any inlets and outlets. As an example, we can have two separate models, one of a gas-liquid separator and one of a simple counter-current heat exchanger. Both models can be simulated by themselves, but can also be connected, creating a new higher-level block to be simulated. This concept of seeing all underlying nodes and arcs in a system as a group is the same concept we used to create the instantiating scheme. If the assumption that the circular nodes have the same properties, then the nodes can be arranged into groups. When applying this concept to the topology of the distillation column, two group nodes can be constructed. One for the liquid parts of the trays and one for gas systems. The resulting higher-level topology is depicted in Figure (1.2). By assuming all the trays in the distillation column

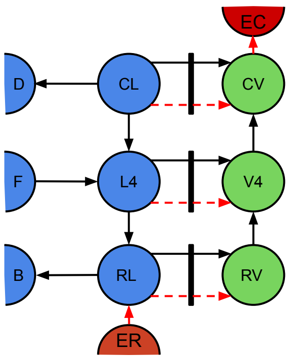


Figure 1.2: Topology of a distillation column with the tray section grouped into one node for the liquid nodes and one for the gas nodes.

have the same properties, the modeler can then to initialize the model on the level best suited for the simulation. Implementing this concept into the instantiating of the models is the main objective of this thesis.

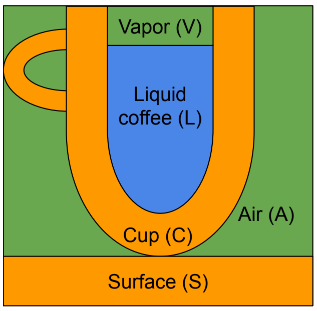
The second objective for this thesis was to make initialization of the models more intuitive. For an initial value problem, the initial states have to be given by the modeler. Typical states for a chemical system is the number of moles and some form of energy, such as enthalpy. Giving the initial value for the states is not, in all cases, an intuitive task. The modeler already has to calculate them from other variables, such as mass, temperature, and pressure. Such variables are often monitored in the plants and systems that the model shall represent and is, therefore, more accessible for the modeler. The mathematical model of the system already contains equations with the possibility of calculating the initial states if they are properly restructured and given a complete set of variables. Checking if the initial states can be calculated by a variable set provided by the modeler, and creating the mathematical framework, if the variable set is complete, will be the second topic of this thesis.

Chapter 2

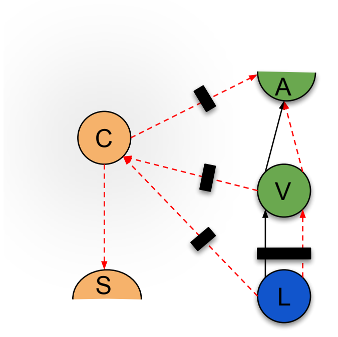
Theoretical fundation

2.1 Topology

The structural component in the model is called a topology. It is a graphical network representation depicting the different entities of the model and how they are connected. It represents the structural part of the model. An example of the topology of a cup of coffee is presented in Figure (2.1b). The two



(a)



(b)

Figure 2.1: Simple topology of a cup of coffee. Some of the hot liquid in the cup will evaporate and be transported to the vapor layer above the liquid before going further to the surrounding air. The heat will be transported from the liquid and vapor to the cup, which will transport it to the surrounding air and the surface the cup sits on.

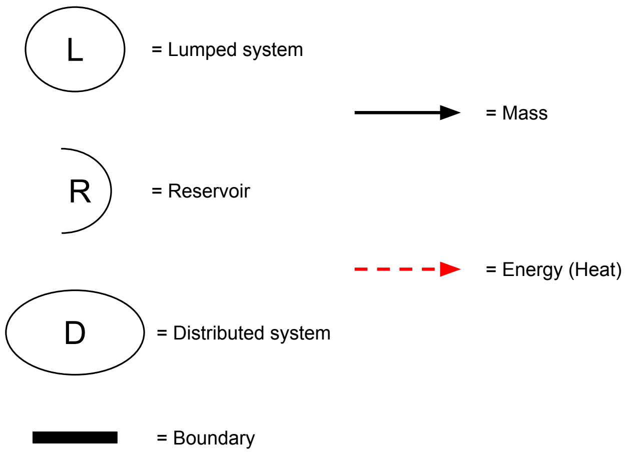


Figure 2.2: The most common arcs and nodes used for graph-based models of chemical systems.

main components of a topology are the nodes and the arcs. The nodes represent individual control volumes containing conserved quantities, while the arcs represent the transport of these quantities between the nodes. These quantities, ”living” and ”moving” in these networks, are called tokens, a term that has been borrowed from petri nets as described in (Petri, 1966). For chemical models, these tokens are typically mass and energy. There are several different types of nodes and arcs, with the most important ones shown in Figure (2.2). The circular nodes are called lumped systems. A lumped system is characterized by assuming a uniform distribution of the intensive properties, such as temperature and pressure, meaning the system is only dependent with regards to time and not any spatial variable. The oval nodes, named distributed systems, are in contrast to lumped systems dependent on both time and spatial coordinates. The introduction of dependency on the spatial coordinates also introduces PDE’s into the model, while a model with only lumped systems only needs ODE’s. PDE’s adds another level of complexity that makes the simulation of the model significantly harder and potentially more time-consuming. It is therefore beneficial to avoid using distributed systems unless the spatial dependency is paramount to the model. A method often used to avoid distributed systems is to model it as a series of lumped systems. The last important type of node is the reservoir, depicted with a half circle. A reservoir is a node thought of to be so much bigger than the order nodes that any transport between the reservoir and a lumped or distributed system is having a negligible effect on the properties in the reservoir. For instance, if a glass with hot water is set in the middle of a room. There will be a transport of both energy and mass from the water to the air in the room, but neither the temperature nor the mass the of the air in the room will change significantly and therefore is assumed constant. Moreover, the reservoirs also set the standards for the driving forces in the model. Meaning that if the mass transfer in the model is only determined by the pressure difference between the nodes, there will ultimately be no transport of mass if the pressure is set equal in all the reservoirs.

As for the arcs, the most important ones, used for chemical models, are shown in Figure (2.2). The labels given to the arcs are specific to what types of models that are being developed. In our case, this is going to be chemical models, hence the labels of *mass* and *energy*. The whole-drawn black arrow represents mass transport. In the models used as examples in this project, this is usually pressure driven mass transport or diffusion driven by a difference in chemical potential. The red dotted arcs account for energy transport through heat, with conductivity or convection (Geankopolis, 2003). The direction of the arrows indicates what the positive direction of the flow is. This does not exclude the possibility of the flow of any token in the opposite direction.

In the topology, there are several different systems with all the systems being divided into zones using different background colors on the nodes, depending on the phase of the system. One section for the gas phase, one for liquid and one for solid phase. The sections are divided by black lines as markings for phase boundaries, for which the token must travel across. In this model, all the system are assumed to be lumped or reservoirs for a more simple example.

Once the topology is set, the incidence matrices can be constructed. These matrices mathematically represent the directionality of the flow of tokens between the nodes in the system. If the node in question is a source node, it is denoted by a 1 and -1 if it is a sink node. With the tokens for the model of a cup with water is mass and energy, these incidence matrices can be constructed.

Table 2.1: Incidence matrix for mass transfer in the system.

	L|V	V|A
L	-1	0
V	1	-1
A	0	1
C	0	0
S	0	0

Table 2.2: Incidence matrix for heat transfer in the system.

	L|V	V|A	L|C	V|C	C|A	C|S
L	-1	0	-1	0	0	0
V	1	-1	0	-1	0	0
A	0	1	0	0	1	0
C	0	0	1	1	-1	-1
S	0	0	0	0	0	1

2.2 Ontology

The *ProcessModellerSuite* is a multi-discipline modelling tool. The knowledge about the different disciplines is stored in an ontology. The ontology is structured in a tree-like formulation where each branch inherits the behavior from its root. An example of such a tree structure is shown in Figure (2.3). The set of ”rules” that is being defined in the ontology is called taxonomy. The ontology provides one set of ”rules” defining the structural-related model components and one set to capture the mathematical behavior of the individual model components. Most of the terms are defined in the root node of the tree and inherited throughout the entirety of the tree. For each level in the tree, the taxonomy can be extended with new terms that are only inherited to the branches unfolding from that node. After the root node, the tokens are introduced, hence the separation between physical properties and control properties. These sections are referred to as networks. The taxonomy for each network is then sequentially adding more information, refining the taxonomy for the networks. For instance, we define tokens in the physical network, then being defined for all sub-networks of the physical network. As an example, we can use the model example

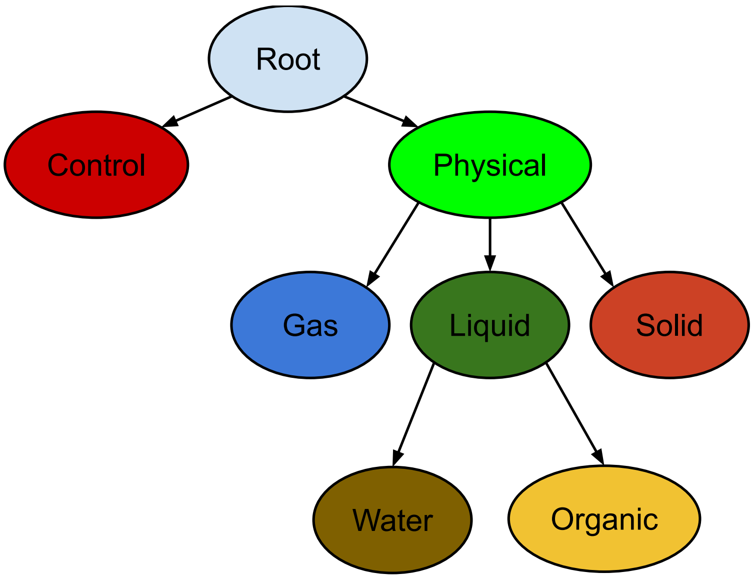


Figure 2.3: Example of a tree-structured ontology.

from the topology section. The tokens defined in the physical network are mass and energy, but there is more than one aggregated state. Three subnetworks for each of the aggregated states are therefore created, whom all inherit the tokens defined. In these nodes, we then add new terms. For instance, there are different ways to calculate the pressure of a system with a gas and a system with liquid. The different transport mechanisms for the tokens are also introduced at this level. The expanding of the network ultimately creates a multi-network ontology. Further description of the concepts of creating ontologies is described in (Preisig and Elve, 2016). For this project, the ontology provides the mathematical framework for which the model has to be modeled in. The possible equations, variable and typed-tokens come from the ontology, along with the dimensionality of said items.

2.3 Index sets

Index set is a concept used by the software to provide information on the mathematical dimensionality of the variables. Each variable has assigned an index set. An index set, such as the *Node set*, $A_{\mathcal{N}}$, would then correspond to the nodes in the model and a variable has the dimensionality of a column vector with the same length as the number of nodes in the system. If a variable has assigned two index sets, such as *Node set* and *Arc set*, $A_{\mathcal{N},\mathcal{A}}$, it has the dimensionality of a matrix with column length indexed by the first index set and row length indexed by the second. Index sets are not limited to correspond with nodes and arcs. For chemical systems, it is typical to introduce species. Species are defined in the software as a *Typed token*. A typed token is a kind of sub-property of a token. Species is an example of a sub-property of mass. A typical index set can then be *Species in node*, $A_{\mathcal{N}\mathcal{S}}$. Since the number, and types, of species in each node, may vary, the dimensionality of such a variable is not trivial. Two nodes with the same number of species and the same types may have a different order of the species in the mathematical description of the node. Therefore, keeping track of the order is also paramount to get the correct calculations. Further elaboration on index sets can be found in (Elve and Preisig, 2019).

As an example, we can use an arbitrary topology as visualized in Figure (2.4). A variable applicable for every node, such as temperature, will have

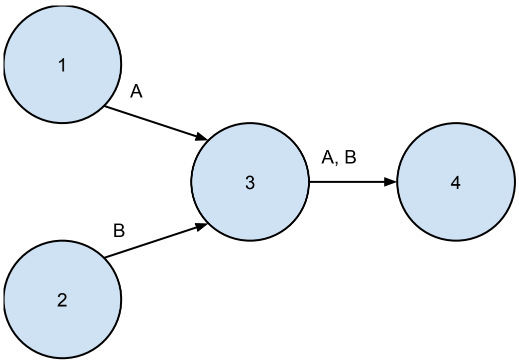


Figure 2.4: Visualization of an arbitrary topology. Species A is transported from 1 to 3 and species B is transported from 2 to 3, before both A and B is transported to 4.

the node index set assigned to itself, $T_{\mathcal{N}}$. Hence, it will be a column vector of length four, as there are for nodes in the system.

$$T_{\mathcal{N}} = \begin{bmatrix} 298 \\ 298 \\ 298 \\ 298 \end{bmatrix}$$

A variable such as the the mass flow incidence matrix, $F^m_{\mathcal{N},\mathcal{A}}$, has to index sets assigned to itself, making it a matrix with the node set giving the dimensionality of the columns and the arc set giving the rows. There are four nodes and three arcs, resulting in a 4x3 matrix.

$$F^m_{\mathcal{N},\mathcal{A}} = \begin{bmatrix} -1 & 0 & 0 \\ 0 & -1 & 0 \\ 1 & 1 & -1 \\ 0 & 0 & 1 \end{bmatrix}$$

2.4 The Grand Scheme

A proper computational framework is a key part of any simulation-driven research. Making sure that we have a complete equation set with the correct order and zero degrees of freedom was one of the tasks for this project. To ensure this, we used the concept of *The Grand Scheme* as proposed in (Preisig, 2010). A visual representation of the concept is presented in Figure (2.5) as a block diagram. The four major blocks that make up the scheme are *balances*, *transport*, *kinetics* and *state variable transformation*. By inserting the initial condition of the states a time equal zero, the time derivatives of the states are calculated. An integrator predicts the values of the states for the next time step and the cycle continues for as long as the modeler wants to. The length of the simulation, as well as the time step, have to be predetermined by the modeler before starting the cycle. As small step sizes give a higher computational accuracy but a larger computational load, the modeler has to do a cost-benefit analysis to determine these settings.

2.4.1 Balances

The balances block is the last block in the computational cycle. This block combines the variables from both the kinetics and the transport block to make the time derivatives. The states can be of both a conserved nature, such as mass and also of a not conserved nature, such as moles when reactions are a part of the model. By taking both of these cases into account, the general term for the time derivatives of the states, or accumulation terms, becomes:

$$\text{Accumulation} = \text{Transport across the boundary} + \text{Internal conversion}$$

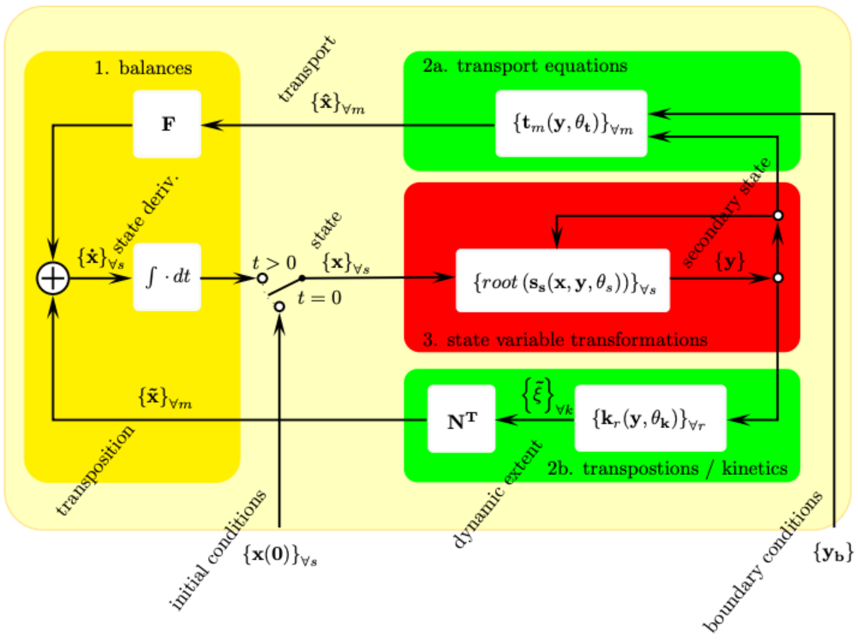

1. balances
F
$\int \cdot dt$
state deriv.
$\{\dot{\mathbf{x}}\}_{\forall s}$
$\{\tilde{\mathbf{x}}\}_{\forall m}$
transposition
transport
$\{\hat{\mathbf{x}}\}_{\forall m}$
2a. transport equations
$\{\mathbf{t}_m(\mathbf{y},\theta_t)\}_{\forall m}$
state
$t>0$
$\{\mathbf{x}\}_{\forall s}$
$t=0$
$\{root\,(\mathbf{s_s}(\mathbf{x},\mathbf{y},\theta_s))\}_{\forall s}$
secondary state
$\{\mathbf{y}\}$
3. state variable transformations
$\mathbf{N}^{\mathbf{T}}$
$\{\tilde{\xi}\}_{\forall k}$
$\{\mathbf{k}_r(\mathbf{y},\theta_k)\}_{\forall r}$
2b. transpostions / kinetics
dynamic extent
initial conditions
$\{\mathbf{x}(0)\}_{\forall s}$
boundary conditions
$\{\mathbf{y_b}\}$

Figure 2.5: *The Grand Scheme* presented as a block diagram showing the different blocks and how they are connected as described in (Preisig, 2016).

These terms are, in turn, inserted into the integrator. The integrator updates the states before the next cycle starts. Many possible integrator schemes can be used. It all again comes down to the evaluation between accuracy and efficiency. The more efficient schemes do the update in one step, like the explicit Euler scheme (Hanna, 1988), while some schemes use intermediate calculations to increase accuracy, such as higher order Runge-Kutta schemes(Butcher, 1996).

2.4.2 Transport

The transport box holds the equations used to calculate the transport between the nodes, represented in the system as arcs. The difference in effort variables, such as pressure, temperature, and chemical potential, drives the transport between adjacent systems who share boundaries. All the effort variables are functions of the derivatives of the states (Callen, 1985). Other factors, such as valves can also manipulate the flows between the systems. The reservoirs set the boundary conditions. The nodes in the system will stabilize to a gradient between the reservoir in the system depending on the difference in the effort variables in said reservoirs.

2.4.3 Kinetics

The kinetics box accounts for the internal conversion in the accumulation term, usually as a reaction between species in chemical systems. The conversion in systems with reactions goes in ratios. These ratios are described by using the stoichiometric coefficients of the species in the reaction. As for the dynamics, they are described using the change in the extent of reaction based on the empirical relation stating that the species physically have to meet for the reaction to occur and that there is a sufficient amount of energy (Atkins and de Paula, 2006).

2.4.4 State variable transformation

The state variable block links the states and the variables in all the other blocks. The equations in this block are also known as the *closure* equations as they close the equation set. These variables, linking the blocks, are termed secondary states. Often these secondary states are the variables we would like to observe. Some examples of such variables are concentration, pressure, and temperature. Creating this block is often the hardest part of the modeling. The relationships between the primary states and the secondary states can potentially be quite complex. Equations of state or partial differential equations of the fundamental energy functions with respect to the extensive quantities are some examples of such complex relationships (Haug-Warberg, 2006). Some relationships are also simple empirical relations, such as volume being the relationship between moles and concentration and density is the relationship between mass and volume.

Chapter 3

Implementation

Before diving into the work done in this thesis, a little clarification on the difference between the words instantiating and initializing may be in order, as they are quite frequently used and may be thought of having the same meaning. In this thesis, instantiating and initializing can be thought of as the qualitative and quantitative parts of the program, respectively. Instantiating is the structuring of the information provided by the various information sources (Decker and Mendling, 2009), while initializing is the assigning of values for states and variables (Esche et al., 2018).

The main objectives for this thesis were to create an instantiating procedure for the graph based models constructed by the *ProcessModellerSuite* (Elve and Preisig, 2019). The scheme was programmed as a python class. By making it as a class, the program is easily implemented into the *ProcessModellerSuite* by importing the class. Creating an instantiating object ensures a proper data structure that the class functions can operate on regardless of changes in the structure in the rest of the software. Moreover, it creates a possibility of different schemes based on what intermediate steps the modeler wants to use to reach to the final initialized model. As a block box model the general scheme would look something like Figure (3.1) From the ontology, chosen by the modeler, the program receives a set of equations, a set of variables and the possible typed tokens for this ontology. These sets give the possibilities that can be used to create the mathematical model of the system. The states give the ”starting points” from where we need to build the mathematical model and make sure the loop comes

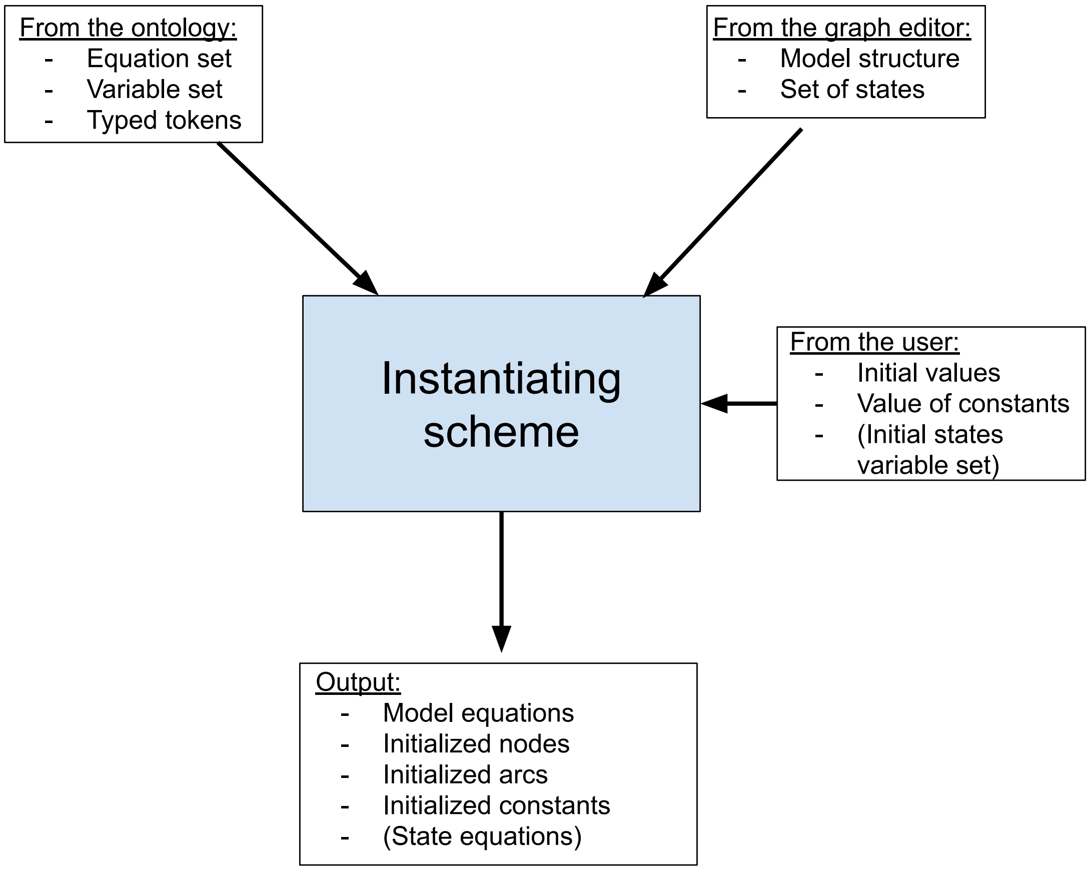


Figure 3.1: Illustration of how the instantiating scheme is working as a black box model.

back around to, as the equation system need to have zero degrees of freedom. The information on the dimensionality of each variable and constant comes from the model structure, as well as information about the grouping of nodes and arcs. The initial values of the states and the values of the constants, needed for the calculation of the model equations, are given as inputs by the user when the scheme is executing. Setting these values can be done in different ways, with the easiest way being to set it for each node and arc separately. We wanted to be able to utilize the grouping structures in the model structure to initialize more than one node and arc at the time and thereby reducing the time needed for initialization. By defining groups for both arcs and nodes, this was made possible.

The output needed to make the simulation program in the *ProcessModeller-Suite* is the model equations in the correct computational order, an overview over all the nodes and arcs with their respective constants initialized and the initial conditions for the states. The simulation also needs network information, such as incidence matrices, and selection matrices, but they are handled elsewhere. The ”state equations” are put in parenthesis because they are optional. They represent an equation set to calculate the initial states. These equations require a variable set with all the variables need to compute one or more states by using a subset of equations from the model equations set. This procedure is not something the modeler has to use but will have the possibility to use. Combining all these tasks gives the algorithm presented in Figure (3.2). The overall algorithm shows the pathway of the information and what choices the modeler has along the way. The following sections will describe each of the boxes function and implementation in detail.

The shape of the different boxes in the algorithm scheme provides separate uses. A circular box indicates information, in the form of variables or boolean statements. These boxes are either, or both, inputs and return values that are being sent in to or extracted from the rectangles. The rectangular boxes represent functions. They take inputs and use them to return a variable or statement to the user. The diamond boxes are logic boxes where a binary question is asked. Depending on the answer, which is always yes/no or true/false, the algorithm continues in the direction with the answer to the question. The information travels along with the arrows, which are uni-directional.

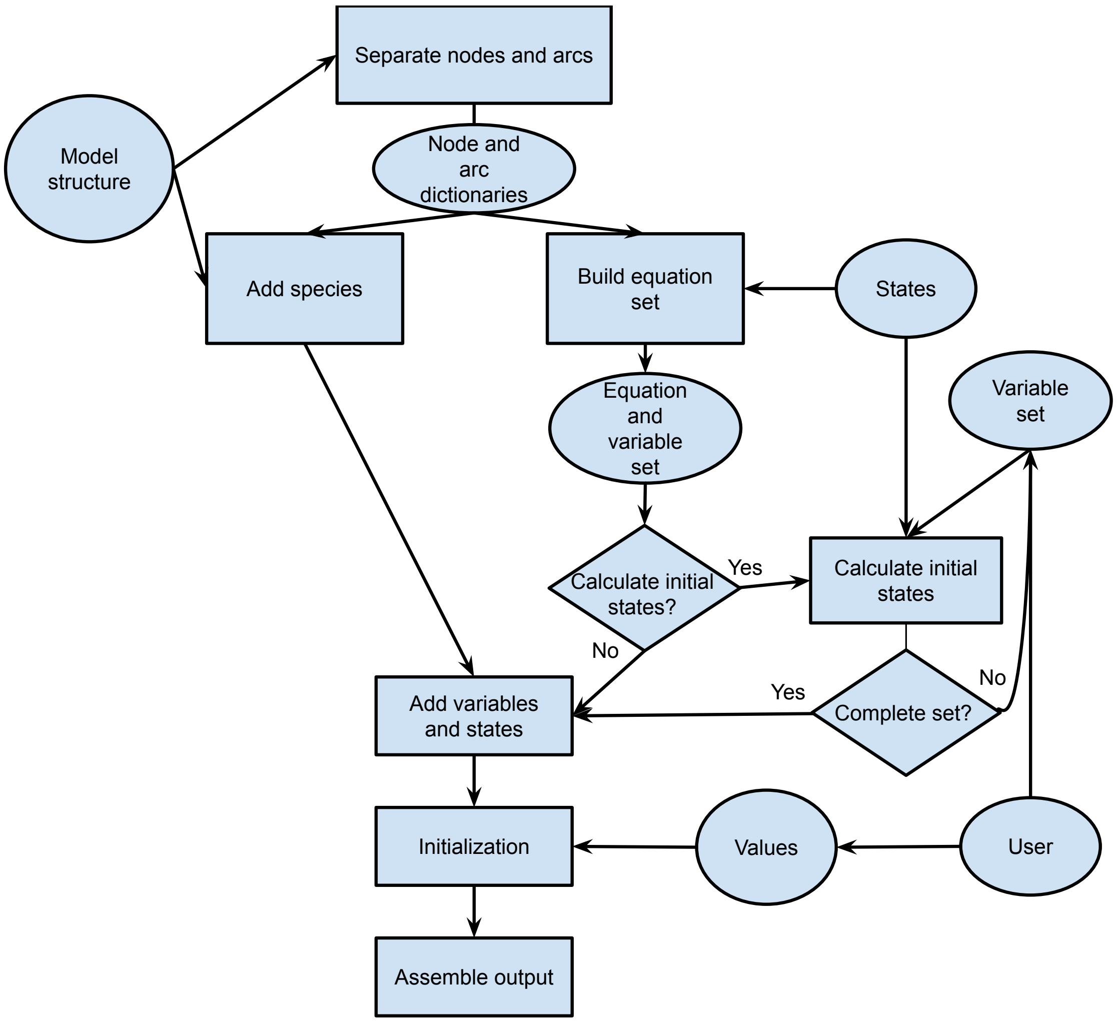


Figure 3.2: An overall algorithm of how the entire instantiating class works.

3.1 Building the class

3.1.1 Structuring the nodes and arcs

The first task of the class is to structure the information coming from the model structure file from the graph editor and properly store it. This process is executed when the instantiating object is created. Dictionaries were used for storing all the information as well as used for the outputs. Dictionaries have the advantage of being able to store different data types and can hold multiple layers of data with easy to use searching algorithms for needed data.

The program creates three separate dictionaries for nodes and three dictionaries for arcs. For nodes, the categories are reservoirs, group nodes, and single nodes. First, the program asses which nodes that are group nodes and which nodes that are a child node of this group. There are no restrictions on having multiple layers of group nodes. A recursion function assesses if there are any group nodes in the list of children for a group node. If there is, the function calls itself to asses how many layers of nodes there are. During this assessment, the function adds every group node to the dictionary creating the opportunity to initialize the group nodes at any level. With the group nodes and their children established, the single nodes and reservoirs can be handled. The structure of the dictionaries for single nodes and reservoirs are the same, but we decided to divide the two to ensure that reservoirs were initialized properly, as they define the boundary conditions of the simulation. For the same reason, reservoirs are not allowed to be part of a group. The function separating the nodes will remove any reservoirs from the set of children in a group node to avoid having multiple reservoirs with the same boundary conditions, resulting in no transport in the system. If the modeler wants to test a system with all the same boundary conditions, these conditions can be set in the initialization phase.

The arcs are divided into single arcs, group arcs inside group nodes, and group arcs that located between group nodes. The software first establishes which arcs that lies in one of the two types of arc groups. For an arc to be part of a group inside a node, both the sink and source node of that arc has to be children of the group node. Arcs that lie between group nodes are sets of arcs that have the same group nodes as sources and sinks. In topologies, it is visualized as a single arc stretching between the nodes, as shown in Figure (3.3b). In reality, are all the arcs from Figure (3.3a) stacked on top of each other, creating the illusion of one group arc. The remaining arcs,

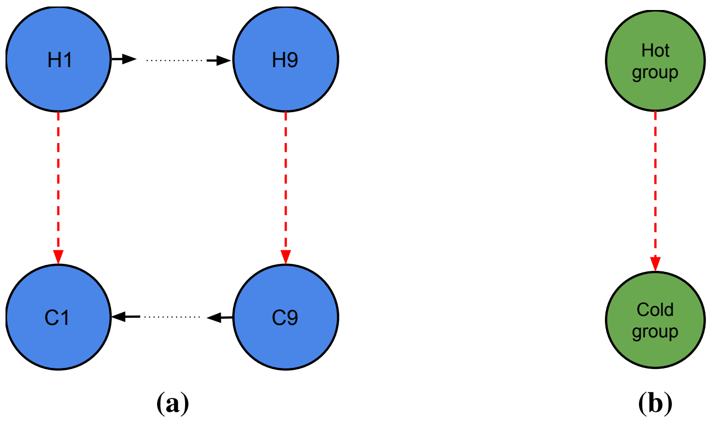


Figure 3.3: An illustration of how two sets of node are being grouped together into two group nodes, and the ”stacking” effect of the arcs going between the nodes inside the group nodes.

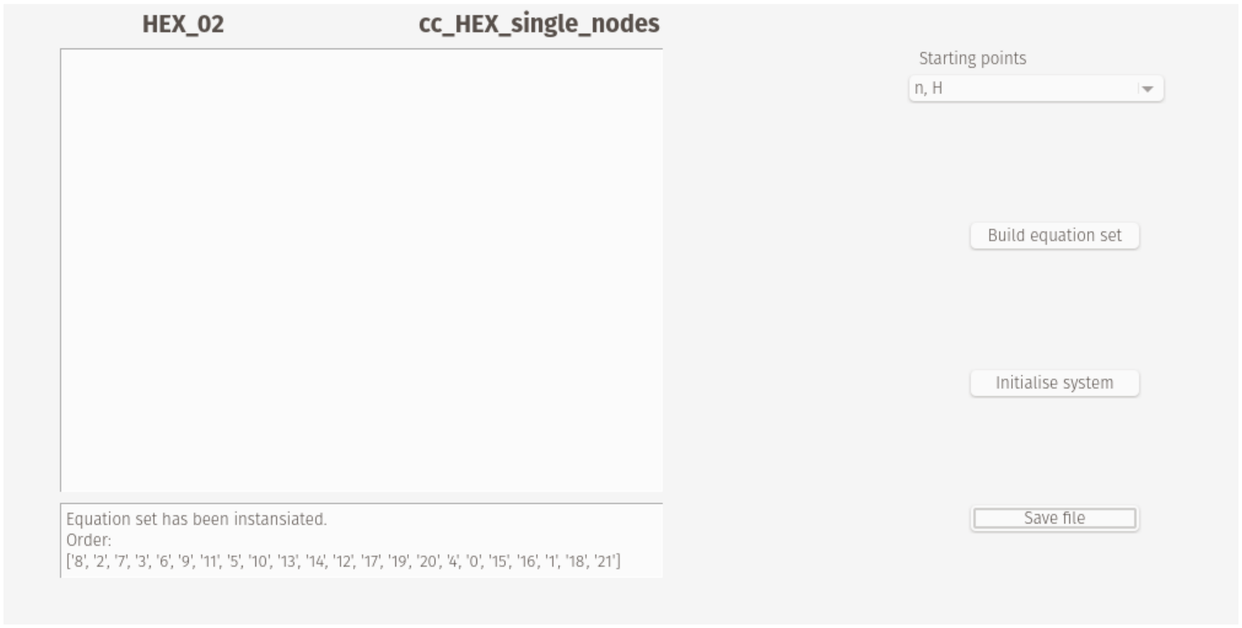
not stored in any of the group dictionaries, are stored in a separate dictionary. The arcs in the group dictionaries are further divided into tokens. The variables are the same for all arcs, but the values of these variables may be quite different. For instance, a typical variable, such as the cross-sectional area. The cross-sectional area for a pipe transporting mass will usually be smaller than the heat transferring area of a heat exchanger. This structure of nodes and arcs is the base that the rest of the program was constructed around.

3.1.2 Adding species and conversions

With the structures for nodes and arcs established, the species and reactions could be added. The information about the species and conversions comes from the ontology as typed-tokens, but the order of the species may differ from system to system and arc to arc. Keeping track of this order is, therefore, paramount to ensure correct calculations in the simulation. For single nodes and arcs, a list with the species was added to their respective dictionaries. If the systems is a mass-based model with no conversions, meaning there are no species, the lists would be empty. The same concept was used for the group nodes and arcs, but as lists within a list were the corresponding list and node, or arc, will have the same index. Also, a list with all the species in the group node was added to get an overview, and to have a more accessible variable to iterate over. The reasoning for this will be discussed further later. The same structure and storage methods were used for the potential conversion that may occur in the nodes.

3.1.3 Building the equation set

Obtaining the mathematical model of the entire system is the overall object of this program. Solving the equations obtained by this program reflect the behavior of the model that is being constructed(Aris, 1978). To get the model equations and constants of the system, the grand scheme, from theory section (2.4), was used. The program requires that the modeler picks the states of the system, which will be the starting point from where the equation set is built. This can be done in a roll-down menu, located in the top right corner in the GUI shown in Figure (3.4). The chosen ontology



```
Equation set has been instansiated.
Order:
['8', '2', '7', '3', '6', '9', '11', '5', '10', '13', '14', '12', '17', '19', '20', '4', '0', '15', '16', '1', '18', '21']
```

Figure 3.4: The graphical user interface where the modeler is instantiating the model and building the equation set.

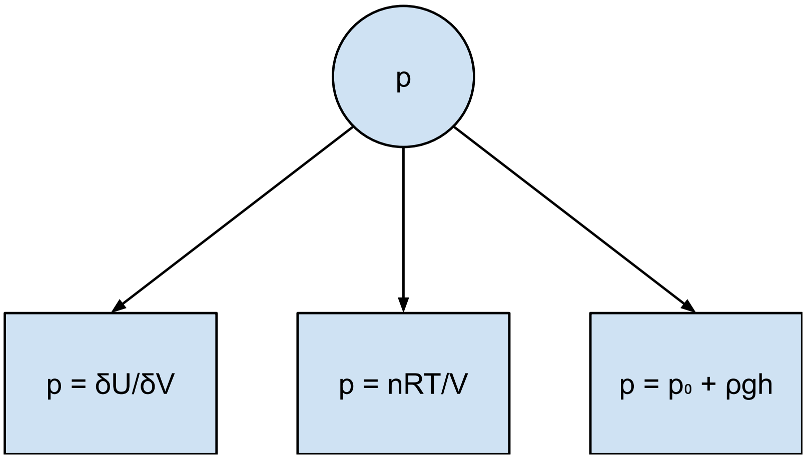


Figure 3.5: Equation options for the calculation on the pressure. All are valid options for calculating the pressure in a node, and gives the same units.

has a finite set of states for models that are constructed under its domain. The menu shows all the possible combinations of states the modeler can choose from. With the starting points in place, the software uses a backward approach of the grand scheme to get the equations. A function evaluates which equation that is needed to compute the state variable. Knowing the equation needed for the state, the function executes a recursive function to ”dive” through the variables and equations, provided by the ontology. Visualization of such a process is present as a flow diagram in Figure (3.6). After the equation for the state has been determined, the recursive function assesses which variables that are needed for the equation. All the variables have an attached list of possible equations that can be used to calculate it. If there is more than one option, a choice has to be made. Ideally, this choice would be taken automatically based on what type of phase that dominates the modeled system. By matching the phase and what networks of the ontology the different equations lie in, a choice could be made on that comparison. When the program was written, all equations were put into the physical layer of the ontology, making it impossible to distinguish between phases. For now, the equations are listed to the modeler, as shown with an example using pressure in Figure (3.5), and the modeler has to choose which equation to use. If the list of equations only contains on possibility, the function continues using this function. When the list is empty, the variable is a constant. The variable can then be a frame variable, such as

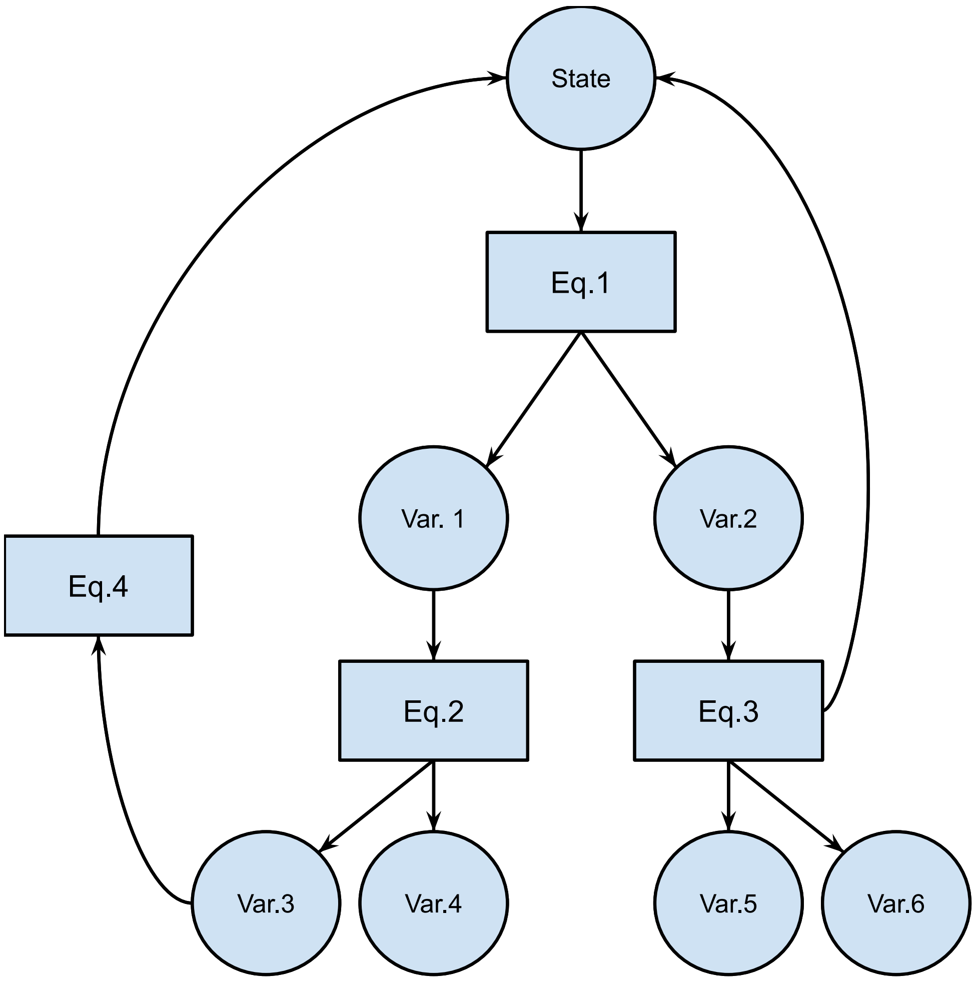


Figure 3.6: A visualization of a simple example of how the equation builder works. By using the state as the starting point, the program works through variables and equations until it reaches a constant or have returned to the state.

time or a spatial variable, a network variable, such as incidence matrices, or constants, termed global variables, such as reference temperature. All of these variables are static, meaning they do not change during the simulation. The constants for frame and network are being handled elsewhere, and are ignored by the program. The global variables, which are the constants the modeler needs to set before the simulation, are kept track of in a list. All the variables that the function examines tracked in a list to ensure that the equation used to calculate this variable only appears once. The recursive function then runs until it hit equations only dependent on constants and states. For the model not to be static, only dependent on set constants, one of the variables has to depend on each of the states to be updated every computational cycle. The ontology takes care of this requirement, so the program does not need to check the connection. When the function terminates, the results are a list of constants that need to be assigned a value and a list of equations that needs to be organized incorrect computational order.

Another issue we encountered was the problem of implicit equation systems. If a variable is needed in an equation that calculates a variable which in turn is going to calculate the original variable, then we have an implicit system, as depicted in Figure (3.7). When this set is being solved, equation 4 need both the state and variable 1 to be known before it can calculate variable 3, but variable 1 is not calculated at that point. Variable 1 is calculated using variable 3 and variable 4. When the program builds the equation set, this creates a loop going from variable 1 to equation 2 to variable 3 to equation 4 and back again to variable 1. Our problem is then recognizing the implicit relation. The solution became to track the variables in each part of the tree separately. When the program has assessed which variables are needed to calculate the state, it will check these variables one at the time. Meaning that it will close the loop back to the state for variable 1 before doing any checks for variable 2. By tracking which variables that already have been assessed in that branch of the tree, the function can identify an implicit relation if it encounters something it already has encountered in that branch and then gives the same commands as if it had encountered a constant or state. Variable 1 thereby add an additional initial value problem, thus needing an initial value to be set for the variable in the same manner as for the states. When the recursive function then returns to the top of a branch, the set of tracked variables in branches below itself is wiped clean ready for the next branch.

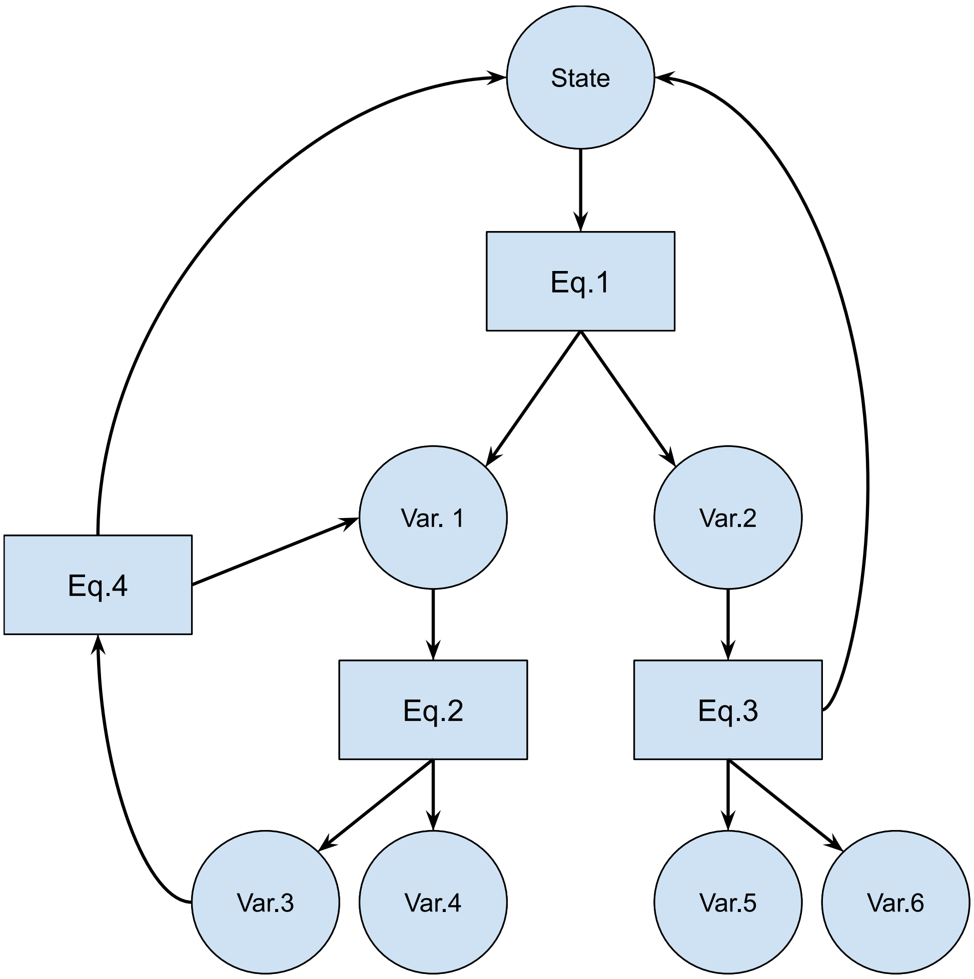


Figure 3.7: The same equation building scheme as shown in Figure (3.6), but with an added implicit relation on variable 1.

3.1.4 Initial state calculation

Setting initial values for the states, such as the number of moles and enthalpy, is not necessarily a trivial task. Process plants are usually not talked of in terms of states, but rather in terms of secondary states, like mole fraction, temperature, and pressure. Often, the initial values already have to be calculated using secondary states. Therefore, one of the objectives of this project was to be able to calculate the initial states of the system by using a set of variables more accessible to modeler or process engineer. Utilizing this part of the class is optional and not called upon unless the modeler wants to.

The set of variables, combined with the constants that are already being tracked in the building of the equation set, is required to span a complete subset of the variables used in the model. To evaluate the possible variable options used to calculate the states, a tree-structured visualization of the dependency of each used variable and state was created. An outline of such a structure is presented in Figure (3.8). The root, top layer, consists of the state. The state then branches out to the variables directly needed to calculate itself. Those variables continue to branch out to the variables needed to calculate them, and so forth. To create this structure, the program uses the same kind of recursion scheme as used to build the equation set with a few modifications. The recursion function runs until the variables needed to calculate a higher level variable is either constants or states. The bottom layer constants and states are not stored in the tree structure as they are going to be set anyways. The function also does not stop if it reaches a variable it has encountered before to visualize all the possibilities for all the states. Moreover, the modeler does not get the chance to pick a different option of equations than what has already been selected in the equation building scheme to keep this scheme within the same mathematical framework.

Once the tree structure is complete, the modeler needs to provide a suggested set of variables to calculate the initial states. If the modeler only wants to calculate the initial values for some of the states, the remaining states must be included in the variable set. The states, not being calculated, can then be used as a variable. By using the suggested variable set, the variable tree for the model, and the states as inputs, the class executes a function using the algorithm presented in Figure (3.10). Since the only incidence, in

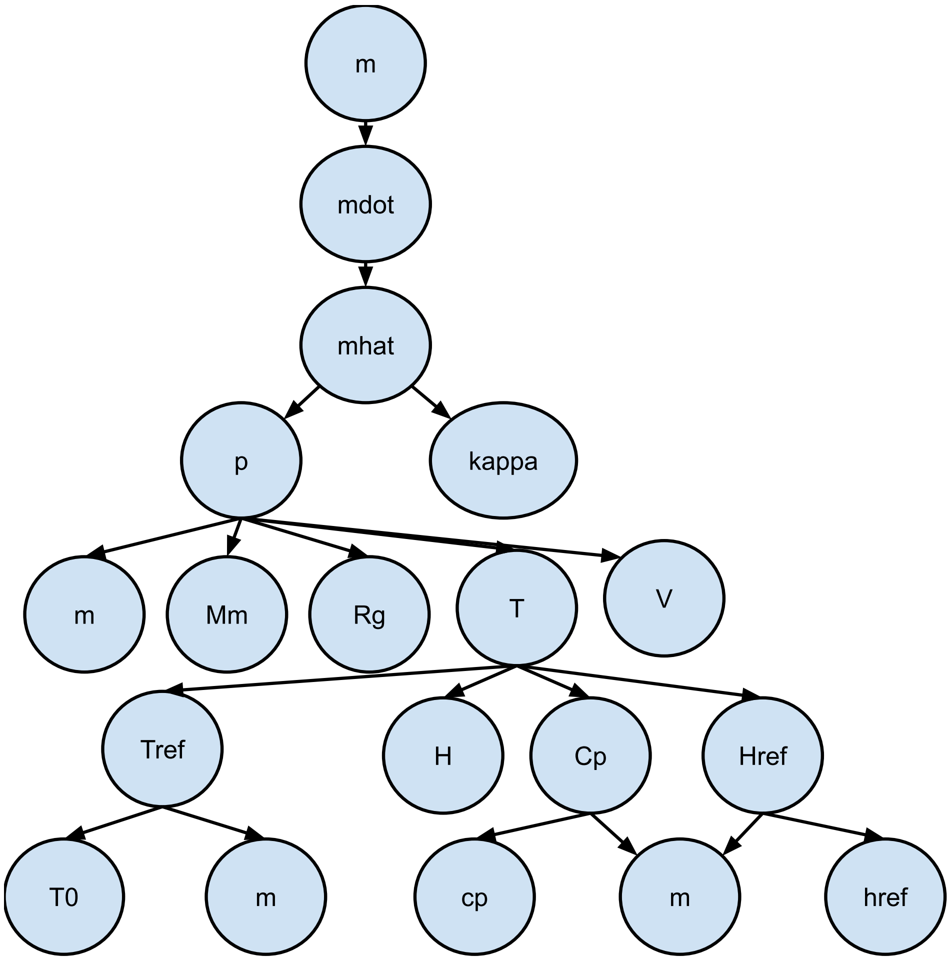


Figure 3.8: Illustration of a variable tree with mass as the state.

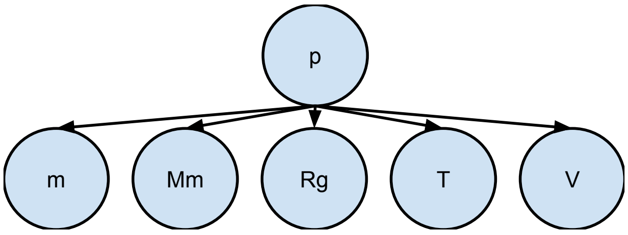


Figure 3.9: A branch of the variable tree showing the calculation of p by using the equation for ideal gas, where the state m is a variable.

the equation set, where a state is calculated explicit as a function of other variables is at the integrator, the program has to find a variable that depends directly on a state and rearrange the equation to be solved for the state. The ”Locate equation and check variable” function locates a branch of the tree where the state in question is used as a variable, as shown by the branch in Figure (3.9), by using the algorithm depicted in Figure (3.11). The function then assesses each of the variables on both the right and left-hand side of the equation. If a variable is in the proposed variable set, previously calculated, or a constant, the function returns a true statement, and the state is added to a list containing all the states that have been calculated. If the left-hand side variable does not meet any of these requirements, the function checks if the variable can be calculated implicitly by using the same scheme used on the states. The right-hand side variables that do not match any of the requirements are checked using an explicit scheme. The explicit scheme evaluates if the variables known to the creates a complete subset of the variables used to calculate the parent variable. If the explicit scheme returns a true statement, the equations needed to calculate the variables are added to a set in the correct order. If not, the function returns a false statement. The right-hand variable should ideally also be check for possible implicit calculations, but trying to achieve this sends the program into an endless loop, which there has been no time to fix.

When a state has been calculated the main function checks all the states that are not calculated once more. This because the calculated states have been added to the variables known to the function. The function assesses if the addition of the calculated state makes it possible to calculate any of the other states. If, in the end, the list containing the calculated states contains all the states, the variables set is complete. The program keeps track

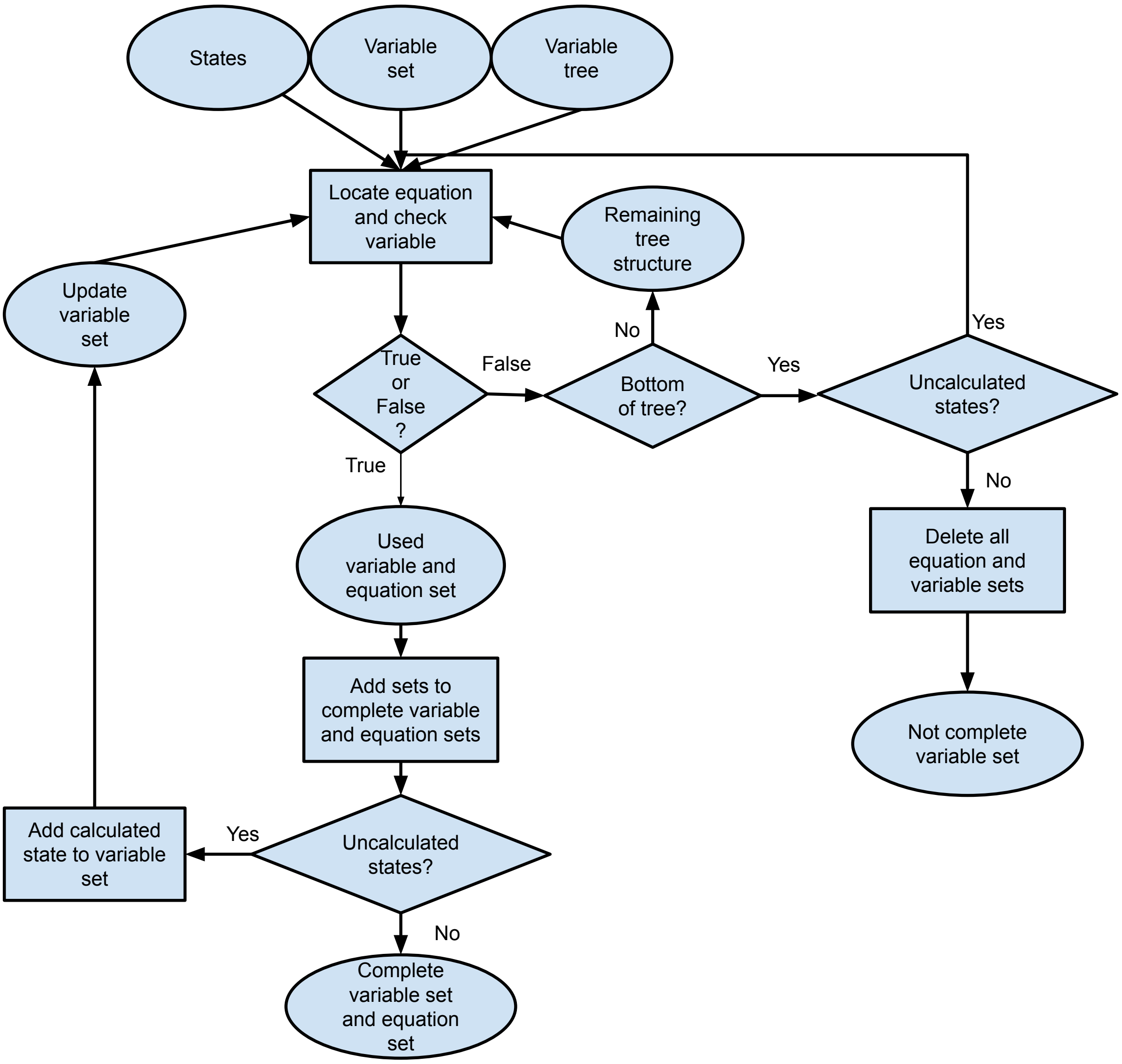

States
Variable set
Variable tree
Locate equation and check variable
Remaining tree structure
Update variable set
True or False ?
False
No
Bottom of tree?
Yes
Yes
Uncalculated states?
True
No
Used variable and equation set
Delete all equation and variable sets
Add sets to complete variable and equation sets
Not complete variable set
Add calculated state to variable set
Yes
Uncalculated states?
No
Complete variable set and equation set

Figure 3.10: A visualization of how the algorithm assesses if a proposed variable set can calculate the initial states by finding solving root functions where the states are variables.

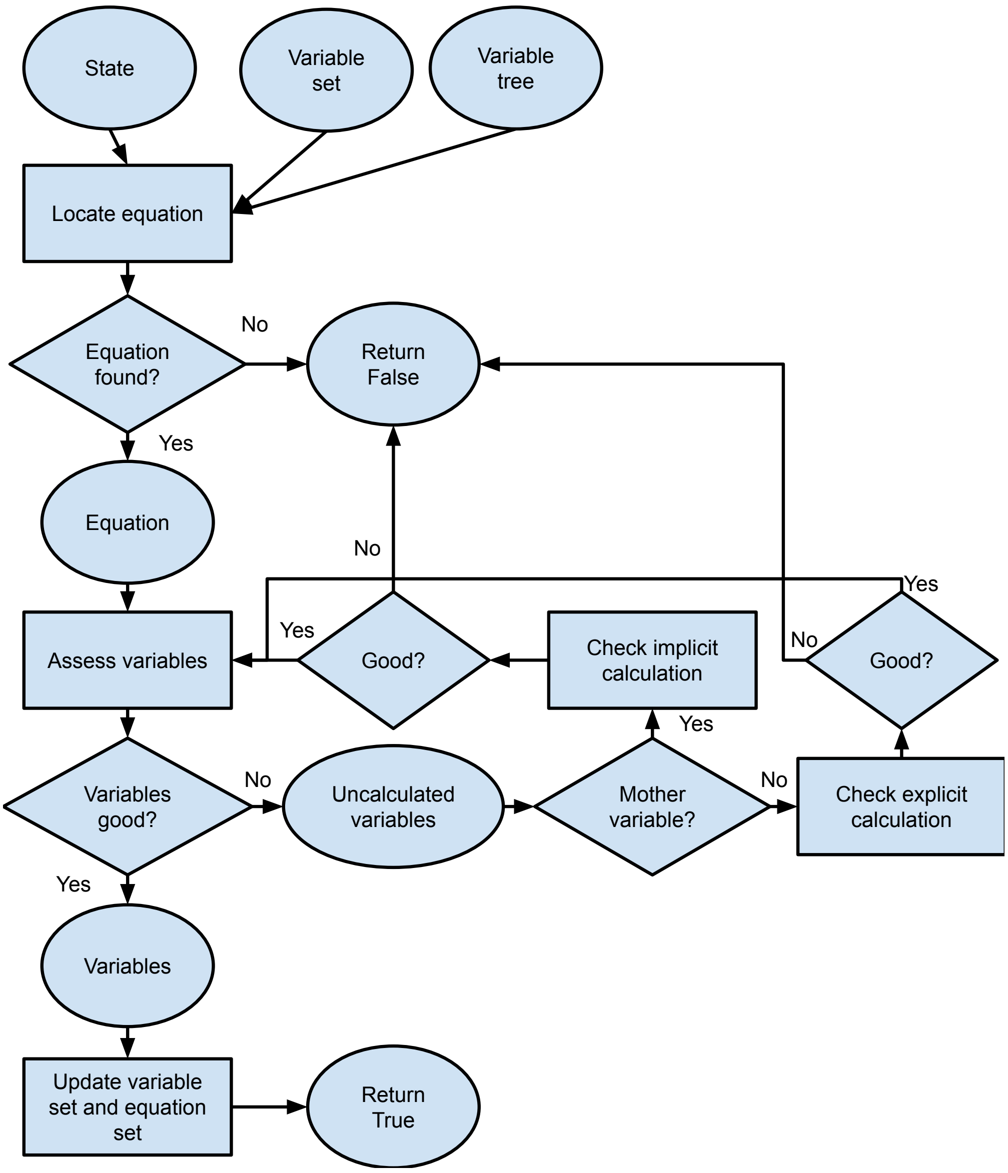


Figure 3.11: The algorithm for the ”Locate equation and check variable” function. The function locates a single equation from the variable tree. If the function can not find an equation, it returns False to the top function. If an equation is found, the function assesses if the variables in the equation are known or can be calculated by known variables. If all variables check out, the variable and equation sets are updated, and the function returns True to the top function. If not the function returns False.

of which variables have been used to make up the complete set. If the recursive function never uses one or more of the variables in the proposed variable set, and the set is complete, the unused variables are discarded from the complete set to avoid unnecessary initialization. If the variable set is not complete, all tracking lists, equation, and variable lists are deleted.

3.1.5 Adding states and constants

The states, constants, and variable set used to calculate the initial states are added to the nodes dictionaries, arcs dictionaries, or a separate dictionary depending on the index set they are assigned. If the variable set is empty, the states are added, but if the set contains variables, they are added instead of the states. All the constants are stored as keys with a column vector attached to it. The length of the vector is dependent on the index set. A constant with index set *node* will have a vector with length one attached to it, while a constant with *arc & species* will have a vector with the same length as the number of species in that arc it is stored in. We opted to use this approach with the constant to keep the nodes and arcs in the center of the initialization scheme. Thus, the functions will iterate through the nodes and arcs, and not through the variables. Constants that are not assigned to nodes or arcs, termed global variables, are stored in a separate dictionary. The constants in this dictionary have index sets related to the typed tokens in the system, such as species and conversions, or no index sets at all. The constant without an index is a scalar, such as the gas constant. The remaining vector dimensions are set by using the information about the type tokens provided by the ontology.

3.1.6 Initialization

With all the mathematical parts of the model sorted, values for all the variables and initial states can be set. The initialization part of the modeling is where we attempt to give the modeler a chance to save time by utilizing the preset groups from the graphical model. Meanwhile, we also want to allow the modeler to use only as much of the group structures as needed, and not limit the program to use all or nothing. All the variables of all the separate entities are initialized to one before the modeler is allowed to set the wanted values to reduce the potential of a system crash when the simulation is running as using the zero as an initial value can cause problems.

The main scheme was broken up into two parts, namely initialization of the entire system and initialization of a single entity. The single initialization is then again dived into functions for all the different possibilities of entities in the system. The modeler chooses a single entity, for instance, a single arc. The program locates to where this arc is stored. This can either be in the single dictionary or one of the two group dictionaries. When the location is known the function iterates over each variable stored in the arc object. If there are more than one species present in the arc, a second iterator sets in and iterates over all the species in the variables with index sets relevant to species. On every instance, the function asks for an input from the user, which in turn is set for that variable. If the arc resides inside a group, the same procedure happens, but the altered values for the single arc does not affect the other arcs in this group. The single entity can also be a group. If a group node is chosen, the procedure more or less the same as for a single node or arc. The difference being that when iterating over the variables and species, the program initializes all the nodes or arcs, who are children of this group, with this value. The list containing all the species, and their order in each child, in the initialized object, ensure that the values are placed in the right place in the value list. Children who already have been initialized are not overwritten. A part of the model that lies on a lower level than the current initialization object cannot be overwritten unless it is reinitialized at the same level it was initialized before. This concept also applies to group nodes that are children of a higher level group node. We assume that when a modeler initializes a part of the model on a lower level, it is because the modeler wants to keep this single entity different from the rest of the group, but still being able to group it.

If the modeler wants to use all the groups at their highest levels, the system-wide initialization scheme can be called. This scheme utilizes many of the same functions as used in the single object initialization. The function iterates over all the nodes and arcs in the system at their highest level. The function can be used as an insurance policy to check if all the parts of the system have been initialized. The modeler can use the single object function at nodes or arcs the needs special attention before executing the system-wide function. The system-wide version will not overwrite any changes made using the single object function. Besides, will the system function also initialize the global variables not subject to any node or arc, as discussed earlier. The initialization function for these variables is written as a separate function being called at the end of the system function.

These initialization schemes would work regardless of the completion of a graphical user interface(GUI). The implementation of the concepts explained over gave GUI-window, as shown in Figure (3.12). The scroll-down menu on the top provides access to all nodes, arcs, groups, and global variables. When selecting an object to initialize all the variables and states are listed, along with the documentation for the variables, as depicted in Figure (3.13). The only column where the modeler can make changes is the far right column. This column contains the vectors with values for the variables. For single entities, the vector is the correct length and denoted with the position each species holds. For a group entity, the window displays a vector with the same length as the number of different species that occur in the group. When the modeler then makes a change, a function updates the values in the chosen entity based on the same principles described above. The same concept of initialization levels applies for the GUI.

3.1.7 Assembling the output and loading cases

The instantiating scheme is not the final product of the *ProcessModeller-Suite*, and the information about the initialized variables and states need to be stored in a convenient way for the other parts of the modeling tool to use. We settle on a format where every node, arc, and global constant is stored separately and allow another part of the modeling software to assemble into the correct vectors for the simulation. Only the single entities are stored to avoid any double information in the output files. The single arcs and nodes that are contained inside groups in the instantiating object are extracted with the vectors referring to the same index in the value lists as the entity has in the list of children. The nodes, arcs, and global variables are stored in separate files under a case name unique to the model.

The cases are stored in a case library in the model repository of the *ProcessModellerSuite*. An instantiating object can load the values stored in the output files into the different entities in the object. This allows the modeler to test cases and make small changes without the need to instantiate the entire model from scratch.

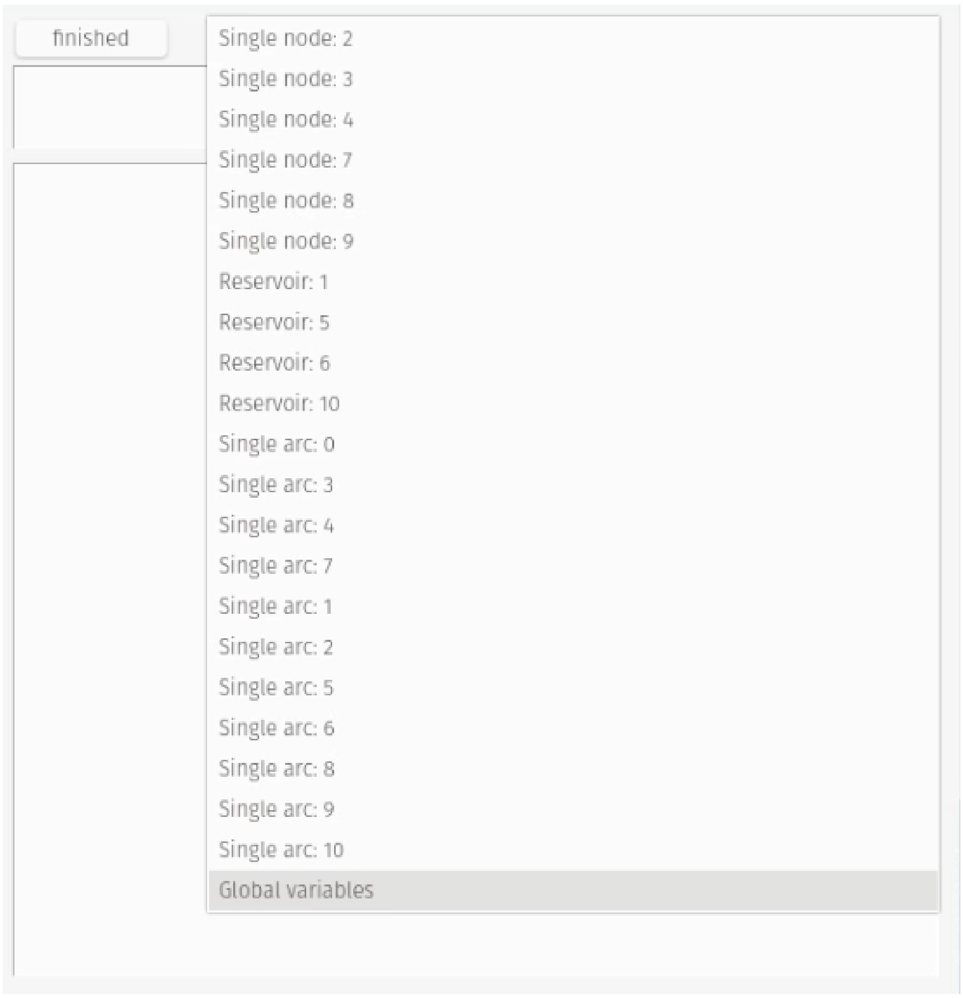
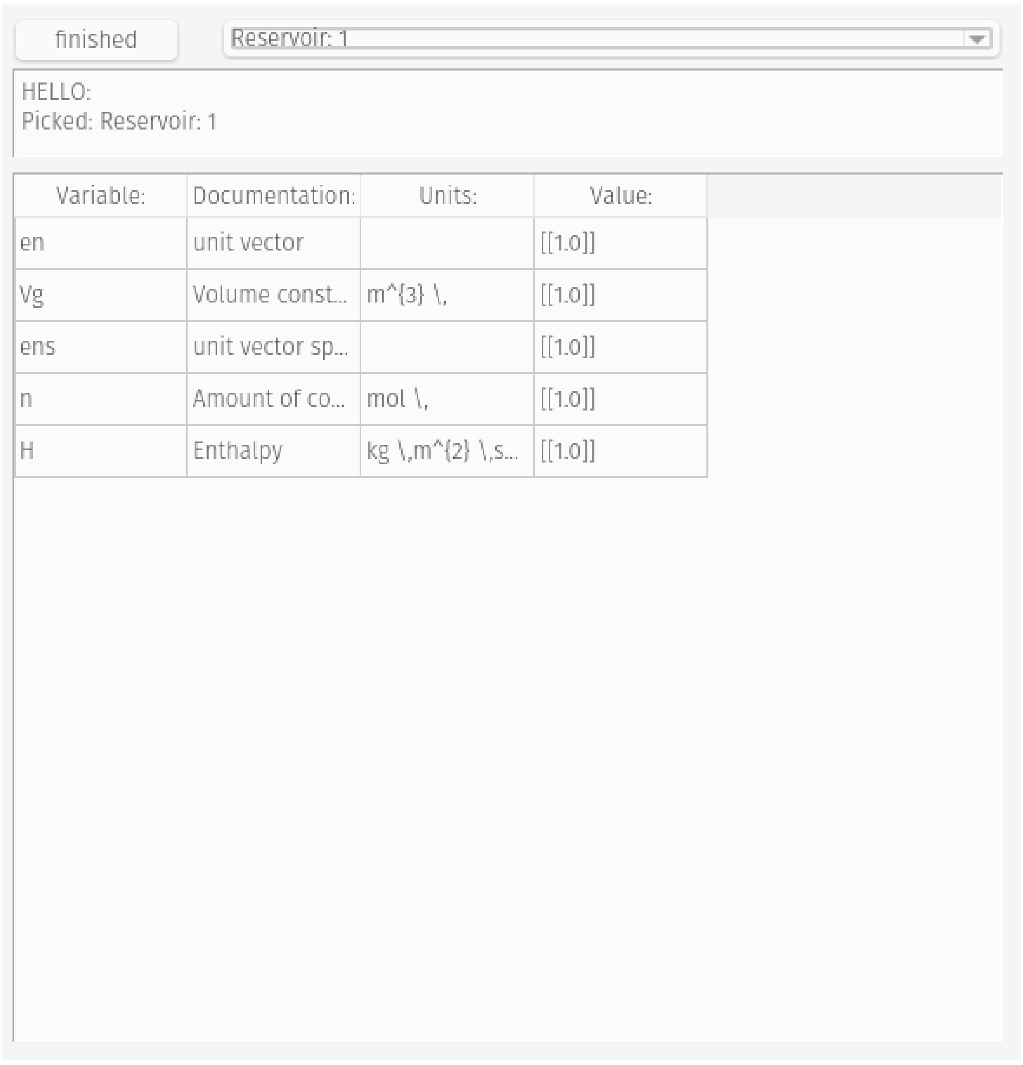


Figure 3.12: The GUI used to initialize the different entities included in the model.



HELLO:
Picked: Reservoir: 1

Variable:	Documentation:	Units:	Value:
en	unit vector		[[1.0]]
Vg	Volume const...	m^{3} \,	[[1.0]]
ens	unit vector sp...		[[1.0]]
n	Amount of co...	mol \,	[[1.0]]
H	Enthalpy	kg \,m^{2} \,s...	[[1.0]]

Figure 3.13: The GUI used to initialize the different entities included in the model

Chapter 4

Simulation of a heat exchanger

To show proof of concept, a couple of models were constructed. The basis for all the models was a simple gas-phase counter-current heat exchanger. A physical layout of the heat exchanger is presented in Figure (4.1). Both the cold and hot side consist of a single chamber with one inlet and outlet and energy being transferred from the hot side to the cold through the wall.

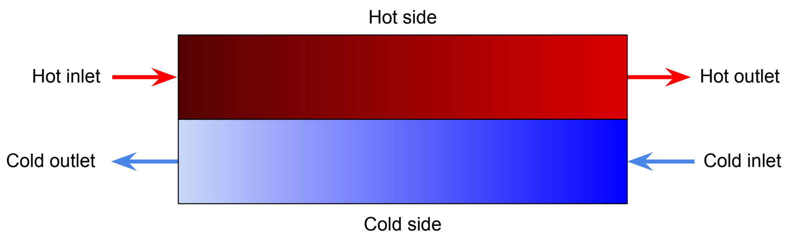


Figure 4.1: A physical outline of a simple counter-current heat exchanger.

4.1 Structural model

By breaking down the physical model of the heat exchanger, a topology, as shown in Figure (4.2), can be constructed. The heat exchanger is modeled with reservoirs for inlets and outlets on both the hot and cold stream. The inside is broken up into N equal lumped systems on the hot side and M

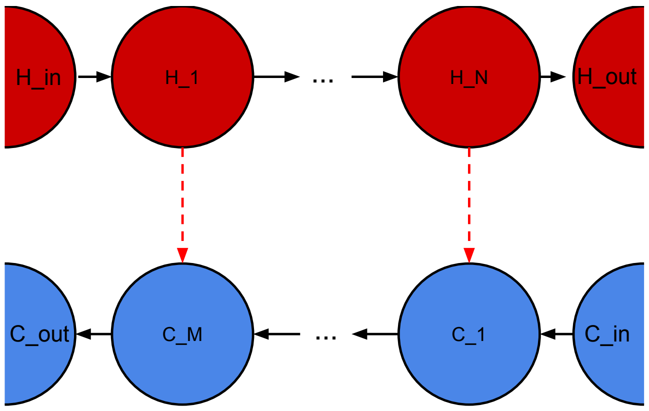


Figure 4.2: A topology of a simple counter-current heat exchanger.

lumped systems on the cold side. With mass transfer between the nodes in the chambers on each side. Heat transfer is limited to occurring between nodes directly across from each other, and any heat loss to the surrounding environment is considered negligible.

4.2 Ontology

To accommodate all the necessary features of the heat exchanger, an ontology, as visualized in Figure (4.3), with the extension of the rules in Figure (4.4), was created. As there was only one phase for this model and no need to differentiate between control and physical properties, the ontology became fairly simple.

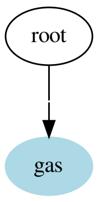


Figure 4.3: Ontology for a gas phase counter-current heat exchanger.

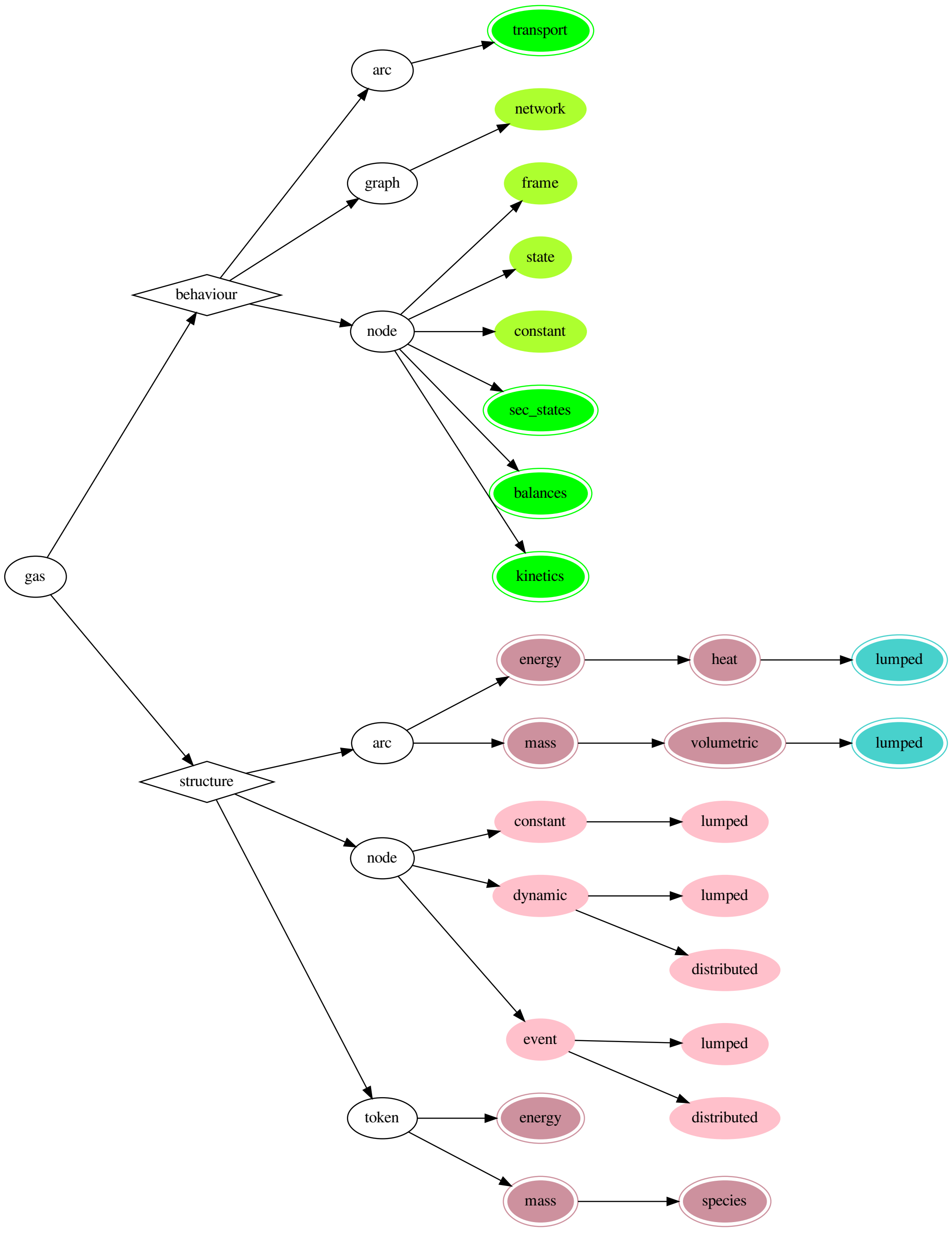


Figure 4.4: The rules for the behavior and structure in the heat exchanger. The nodes highlighted with a circle are rules added in the gas network, while the rest are inherited from the root.

4.3 Mathematical model

By combining the ontology with the topology, and setting the states to be number of moles, *n*, and enthalpy, *H*, the following mathematical model was created by the class. The model will be presented in detail, in the same order as the program builds it, with all variables used listed in Table (4.1) with explanation, as well as the mathematical operators used listed in Table (4.2).

Table 4.1: An overview of all the variables used in the heat exchanger model.

Variable	Documentation	Type
$\dot{n}_{\mathcal{NS}}$	Component balance	Balances
$\dot{H}_{\mathcal{N}}$	Energy balance	Balances
$A_{\mathcal{A}}$	Cross sectional area flow	Constant
$Mm_{\mathcal{S}}$	Molar mass	Constant
R	Gas constant	Constant
T^{298}	Temperature reference	Constant
$T^{ref}{}_{\mathcal{N}}$	Temperature reference	Constant
$U_{\mathcal{A}}$	Heat transfer coefficient	Constant
$V_{\mathcal{N}}$	Volume constant	Constant
$cp_{\mathcal{NS}}$	Heat capacity	Constant
$cp_{\mathcal{S}}$	Heat cap. components	Constant
$e_{\mathcal{N}}$	unit vector	Constant
$e_{\mathcal{NS}}$	unit vector species	Constant
$h^0{}_{\mathcal{S}}$	enthalpy ref	Constant
$h^{ref}{}_{\mathcal{NS}}$	enthalpy ref in nodes	Constant
$\frac{1}{2}$	half, 0.5	Constant
$\kappa_{\mathcal{A}}$	Mass transfer coefficient	Constant
Δt	time step	Frame
t	time	Frame
t^n	integrator end	Frame
t^0	integrator start	Frame
$F^q{}_{\mathcal{N},\mathcal{A}}$	incidence matrix heat flow	Network
$F^m{}_{\mathcal{N},\mathcal{A}}$	incidence matrix mass flow	Network
$F^n{}_{\mathcal{NS},\mathcal{A}}$	incidence matrix	Network
$F^n{}_{\mathcal{NS},\mathcal{AS}}$	incidence matrix	Network
$P_{\mathcal{S},\mathcal{AS}}$	projection matrix	Network
$P_{\mathcal{R},\mathcal{NR}}$	projection matrix	Network

Table 4.1: An overview of all the variables used in the heat exchanger model.

Variable	**Documentation**	**Type**
$P_{\mathcal{S},\mathcal{NS}}$	projection matrix	Network
$P_{\mathcal{NS},\mathcal{AS}}$	projection matrix	Network
${R^N}_{\mathcal{S},\mathcal{R}}$	convertion ratio matrix	Network
$Cp_{\mathcal{N}}$	Heat capacity	Secondary state
${H^{ref}}_{\mathcal{N}}$	Enthalpy reference	Secondary state
$T_{\mathcal{N}}$	Temperature	Secondary state
$c_{\mathcal{NS}}$	concentration	Secondary state
${n^{tot}}_{\mathcal{N}}$	amount of moles in node	Secondary states
$p_{\mathcal{N}}$	pressure	Secondary states
$H_{\mathcal{N}}$	Enthalpy	State
$n_{\mathcal{NS}}$	Amount of component	State
$\hat{H}_{\mathcal{A}}$	Enthalpy flow	Transport
$\hat{V}_{\mathcal{A}}$	Volumetric flow	Transport
$\hat{c}_{\mathcal{AS}}$	Concentration in flow	Transport
$d_{\mathcal{A}}$	direction of flow	Transport
$\hat{n}_{\mathcal{AS}}$	Component flow	Transport
$\hat{q}_{\mathcal{A}}$	Heat flow	Transport
$s_{\mathcal{N},\mathcal{A}}$	Flow node selection	Transport

Table 4.2: Overview of the mathematical operator used in the model with explanation as used in (Elve and Preisig, 2019)

Operator	**Explanation**
$x_{\mathcal{N},\mathcal{A}} \overset{\mathcal{A}}{\star} y_{\mathcal{A}}$	Reduction product over $\mathcal{A}$
$x_{\mathcal{NS}} \odot y_{\mathcal{NS}}$	Kahtri Rao product(Kahtri and Rao, 1968)
$x_{\mathcal{N}} \cdot y_{\mathcal{N}}$	Expansion product
$x_{\mathcal{N}} + y_{\mathcal{N}}$	Addition
$x_{\mathcal{N}} - y_{\mathcal{N}}$	Subtraction
$sign(x_{\mathcal{N}})$	Sign function
$abs(x_{\mathcal{N}})$	Absolute value function
$inv(x_{\mathcal{N}})$	Inverse

4.3.1 Balances

With *n* and *H* as the selected states the time derivatives became as described in Equation (4.1) and (4.2).

$$\dot{n}_{\mathcal{NS}} = {F^n}_{\mathcal{NS},\mathcal{AS}} \overset{\mathcal{AS}}{\star} \hat{n}_{\mathcal{AS}} \tag{4.1}$$

$$\dot{H}_{\mathcal{N}} = \left({F^m}_{\mathcal{N},\mathcal{A}} \overset{\mathcal{A}}{\star} \hat{H}_{\mathcal{A}}\right) + \left({F^q}_{\mathcal{N},\mathcal{A}} \overset{\mathcal{A}}{\star} \hat{q}_{\mathcal{A}}\right) \tag{4.2}$$

Since there are no reactions in the system the changes in the system is only due to transport of components and energy. $\dot{n}$ is only dependent on the mass transport, while $\dot{H}$ is dependent on both the energy transfer that occurs between the hot and cold side through conduction, $\hat{q}$, and the energy that is transferred with the mass, $\hat{H}$.

4.3.2 Transport

The transport of the species through the system was calculated by using Equation (4.3)-(4.7)

$$\hat{n}_{\mathcal{AS}} = \hat{V}_{\mathcal{A}} \odot \hat{c}_{\mathcal{AS}} \tag{4.3}$$

$$\hat{c}_{\mathcal{AS}} = (s_{\mathcal{N},\mathcal{A}} \odot P_{\mathcal{NS},\mathcal{AS}}) \overset{\mathcal{NS}}{\star} c_{\mathcal{NS}} \tag{4.4}$$

$$\hat{V}_{\mathcal{A}} = ((-(\kappa_{\mathcal{A}})) \,.\, (A_{\mathcal{A}})) \,.\, ({F^m}_{\mathcal{N},\mathcal{A}} \overset{\mathcal{N}}{\star} p_{\mathcal{N}}) \tag{4.5}$$

$$s_{\mathcal{N},\mathcal{A}} = (\frac{1}{2}) \,.\, ((abs\,({F^m}_{\mathcal{N},\mathcal{A}})) + (({F^m}_{\mathcal{N},\mathcal{A}}) \,.\, (d_{\mathcal{A}}))) \tag{4.6}$$

$$d_{\mathcal{A}} = sign\left({F^m}_{\mathcal{N},\mathcal{A}} \overset{\mathcal{N}}{\star} p_{\mathcal{N}}\right) \tag{4.7}$$

The flow of the species, $\hat{n}_{\mathcal{AS}}$, is calculated by using the volumetric flow, $\hat{V}_{\mathcal{A}}$, and concentration flow, $\hat{c}_{\mathcal{AS}}$. The volumetric flow is calculated using pressure as the driving force in a linear valve equation. $s_{\mathcal{N},\mathcal{A}}$ gives the direction of the flow. The arrows in the topology indicates the positive flow direction making any flow values with a negative sign flow against the arrows direction. The energy transport was calculated using Equation (4.8) and (4.9).

$$\hat{H}_{\mathcal{A}} = ((inv\left(s_{\mathcal{N},\mathcal{A}} \overset{\mathcal{N}}{\star} V_{\mathcal{N}}\right)) . (\hat{V}_{\mathcal{A}})) . (s_{\mathcal{N},\mathcal{A}} \overset{\mathcal{N}}{\star} H_{\mathcal{N}}) \tag{4.8}$$

$$\hat{q}_{\mathcal{A}} = ((-(U_{\mathcal{A}})) . (A_{\mathcal{A}})) . (F^{q}{}_{\mathcal{N},\mathcal{A}} \overset{\mathcal{N}}{\star} T_{\mathcal{N}}) \tag{4.9}$$

4.3.3 Closure

To close out the equation set, the following secondary states needed to be calculated:

$$p_{\mathcal{N}} = (((inv(V_{\mathcal{N}})) . (n^{tot}{}_{\mathcal{N}})) . (T_{\mathcal{N}})) . (R) \tag{4.10}$$

$$T_{\mathcal{N}} = \left((inv(Cp_{\mathcal{N}})) . (H_{\mathcal{N}} - H^{ref}{}_{\mathcal{N}})\right) + T^{ref}{}_{\mathcal{N}} \tag{4.11}$$

$$T^{ref}{}_{\mathcal{N}} = (e_{\mathcal{N}}) . (T^{298}) \tag{4.12}$$

$$H^{ref}{}_{\mathcal{N}} = h^{ref}{}_{\mathcal{NS}} \overset{\mathcal{S} \in \mathcal{NS}}{\star} n_{\mathcal{NS}} \tag{4.13}$$

$$Cp_{\mathcal{N}} = cp_{\mathcal{NS}} \overset{\mathcal{S} \in \mathcal{NS}}{\star} n_{\mathcal{NS}} \tag{4.14}$$

$$n^{tot}{}_{\mathcal{N}} = e_{\mathcal{NS}} \overset{\mathcal{S} \in \mathcal{NS}}{\star} n_{\mathcal{NS}} \tag{4.15}$$

$$c_{\mathcal{NS}} = inv(V_{\mathcal{N}}) \odot n_{\mathcal{NS}} \tag{4.16}$$

$$cp_{\mathcal{NS}} = cp_{\mathcal{S}} \overset{\mathcal{S}}{\star} P_{\mathcal{S},\mathcal{NS}} \tag{4.17}$$

$$h^{ref}{}_{\mathcal{NS}} = h^{0}{}_{\mathcal{S}} \overset{\mathcal{S}}{\star} P_{\mathcal{S},\mathcal{NS}} \tag{4.18}$$

$$F^{n}{}_{\mathcal{NS},\mathcal{AS}} = F^{m}{}_{\mathcal{N},\mathcal{A}} \odot P_{\mathcal{NS},\mathcal{AS}} \tag{4.19}$$

$$P_{\mathcal{NS},\mathcal{AS}} = P_{\mathcal{S},\mathcal{NS}} \overset{\mathcal{S}}{\star} P_{\mathcal{S},\mathcal{AS}} \tag{4.20}$$

As can be seen from the closure equation, they all end up depending on variables that are of the types network, constant, or state, which are all set by the system or by the user, confirming that there are zero degrees of freedom.

4.4 Cases

To illustrate the possible initialization schemes, a couple of different models with various sizing and groups will be presented in the section.

4.4.1 Case 1: All singles

The simplest version of the model is a model without groups, only single nodes, and reservoirs, as depicted in Figure (4.5). All the nodes and are

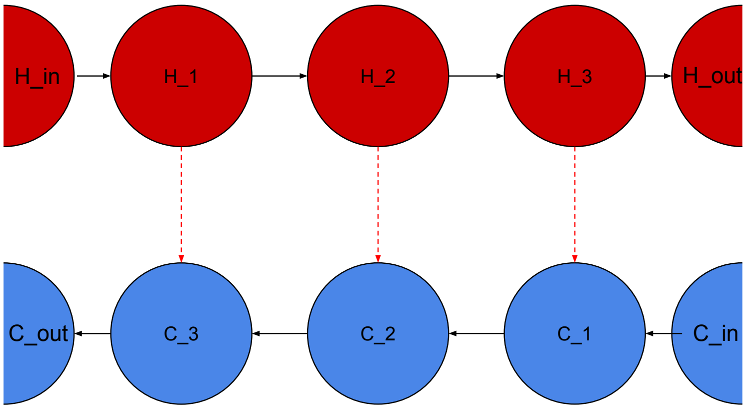


Figure 4.5: Topology of the structural model for case 1.

initialized one by one using the single unit initialization scheme. By setting the nodes one by one, the initial condition is set as a gradient between the two reservoir making the simulation arriving faster at the solution. A stop can then be set when the change in value is lower than the preset value. The results from the simulations can be seen in Figure (4.6) and (4.7). The simulation was performed using a simulation template, as shown in the Appendix. The main simulation template collects the initial values, constants, and network variables from attached files.

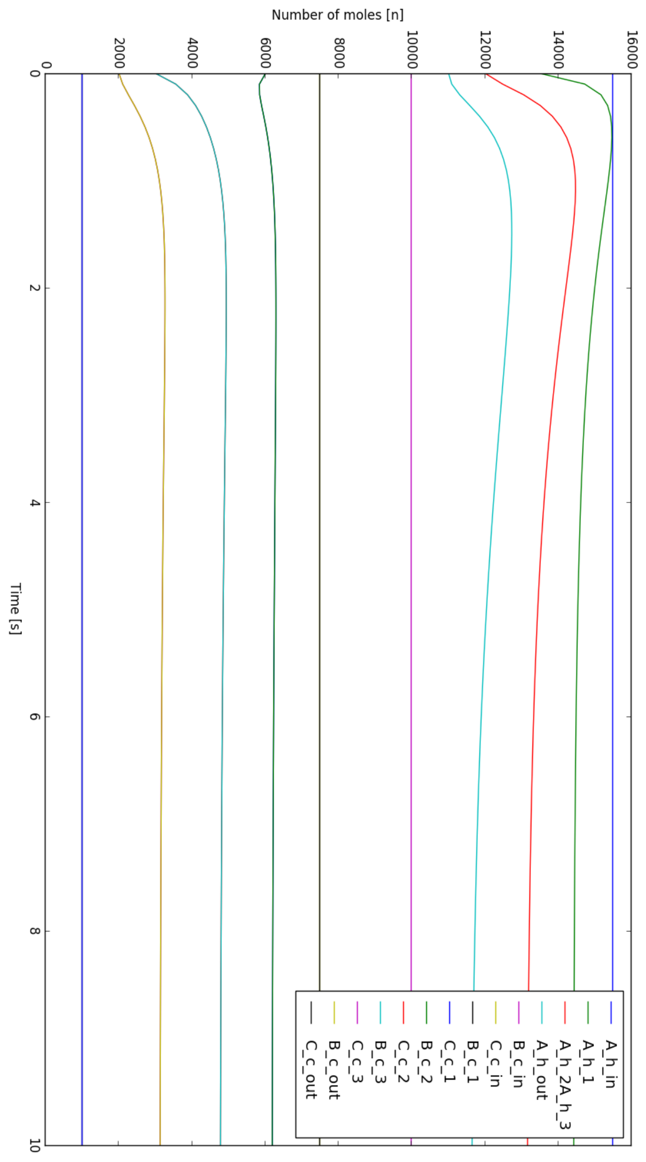


Figure 4.6: Simulation of number of moles in case 1.

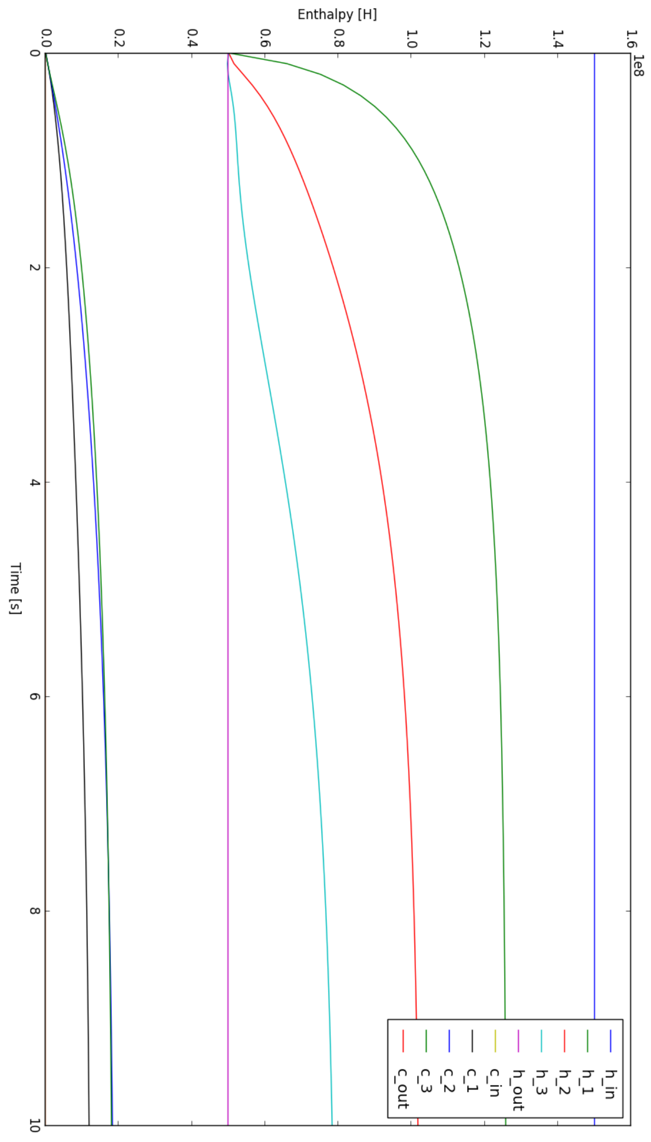


Figure 4.7: Simulation of enthalpy in case 1.

4.4.2 Case 2: Large model

To showcase the main reason the class was built, a model with a substantial number of nodes on each side was made. The lowest level topology is shown in Figure (4.8). To reduce the initialization time all the dynamic

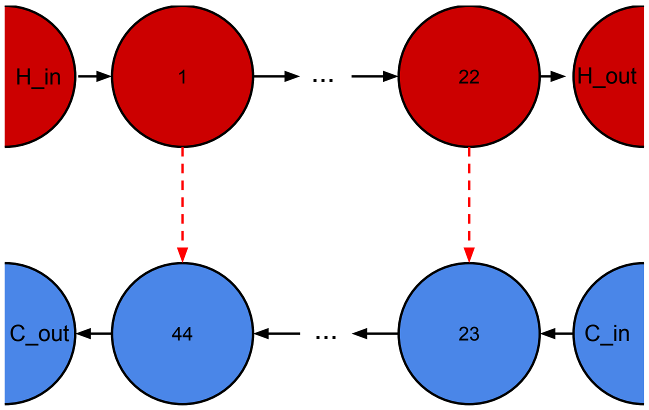


Figure 4.8: Low level topology of the structural model for case 2.

nodes on both the hot and cold side was grouped into one group node, assuming equal properties for all nodes on both sides. This assumption gives the higher level topology shown in Figure (4.9). Using the group structure

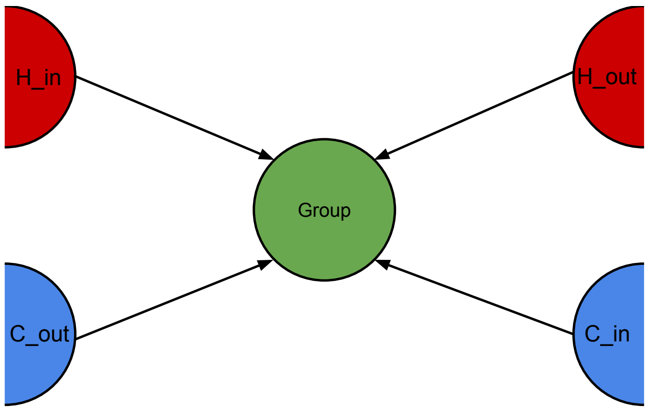


Figure 4.9: High level topology of the structural model for case 2.

in the initialization of the model gives the results depicted in Figure (4.10) and (4.11)

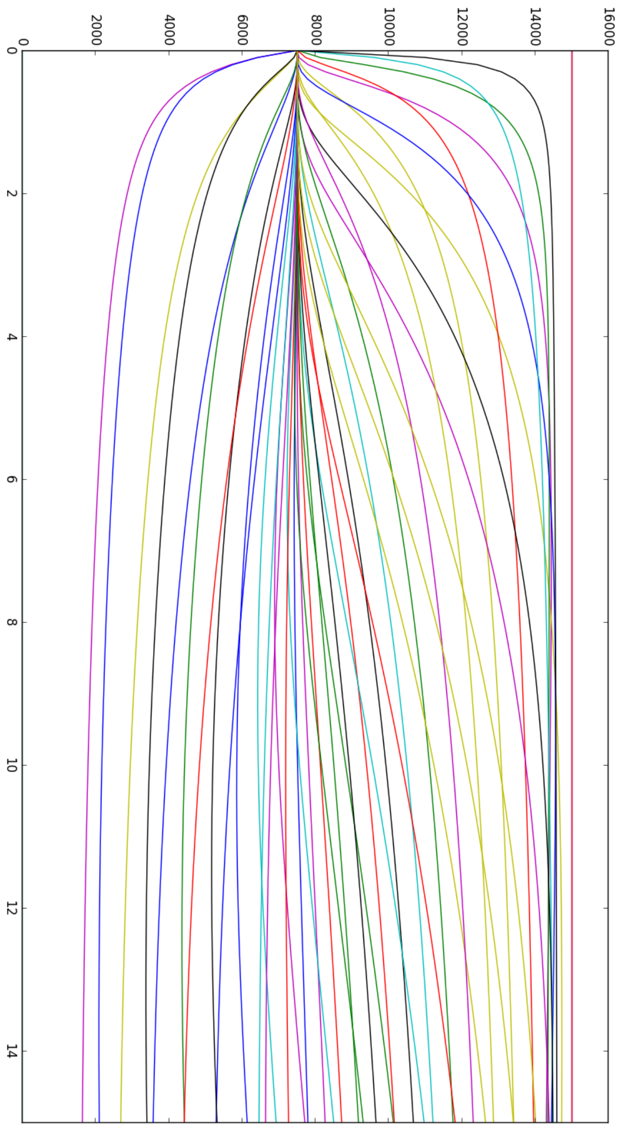


Figure 4.10: Simulation of number of moles in case 1.

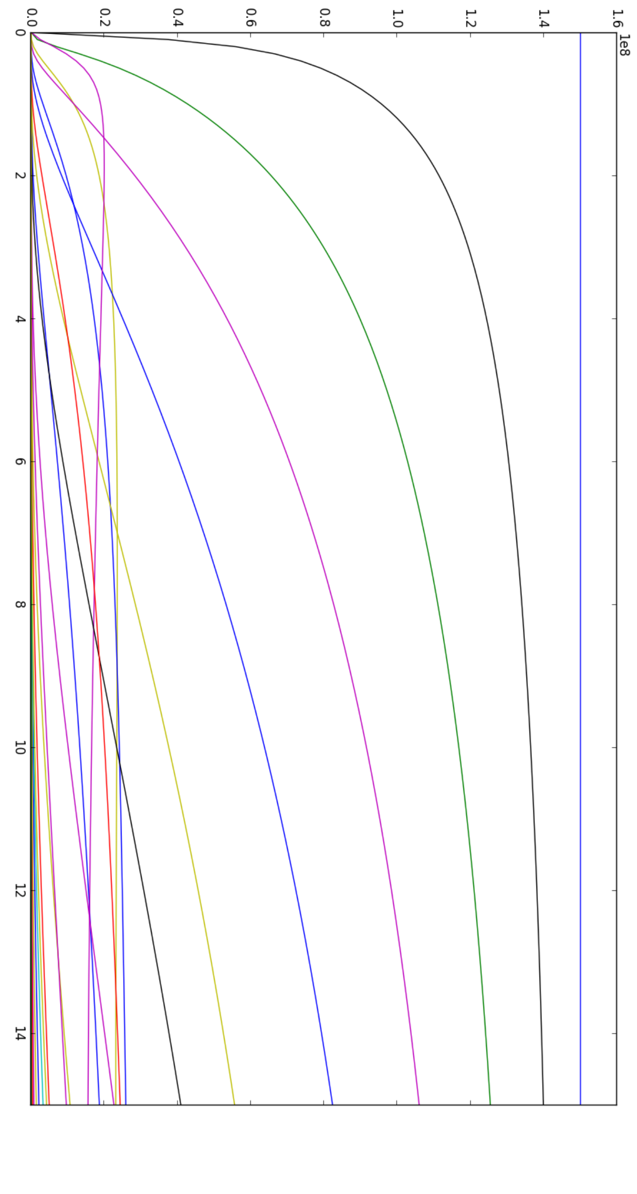


Figure 4.11: Simulation of enthalpy in case 1.

4.5 Initial state calculation

The initial state calculation scheme is not implemented into the *Process-ModellerSuite* so it can not be called from a GUI, but an example can be made from the generated model for this thesis. To calculate the initial states, we chose two secondary states to be used. The temperature in the nodes, $T_{\mathcal{N}}$, and the pressure in the nodes, $p_{\mathcal{N}}$. Combined with the preset constant, these two variables provide enough information to calculate the initial states. The first state the program calculates is $n_{\mathcal{NS}}$ by finding Equation (4.21).

$$n^{tot}{}_{\mathcal{N}} = e_{\mathcal{NS}} \overset{\mathcal{S} \in \mathcal{NS}}{\star} n_{\mathcal{NS}} \tag{4.21}$$

$e_{\mathcal{NS}}$ is a constant already set in the system, but $n^{tot}_{\mathcal{N}}$ is not known to the system. $n^{tot}_{\mathcal{N}}$ is therfore checked for implicit calculation options finding Equation (4.22).

$$p_{\mathcal{N}} = (((inv\,(V_{\mathcal{N}}))\,.\,(n^{tot}{}_{\mathcal{N}}))\,.\,(T_{\mathcal{N}}))\,.\,(R) \tag{4.22}$$

All the variables in this equation, except for $n^{tot}_{\mathcal{N}}$, is know to the system, making it possible to calculate $n^{tot}_{\mathcal{N}}$ by rearranging the equation, and thus be able to calculate $n_{\mathcal{NS}}$. The equation with the possibility to calculate H_N is Equation (4.23).

$$T_{\mathcal{N}} = \big((inv\,(Cp_{\mathcal{N}}))\,.\,(H_{\mathcal{N}} - H^{ref}{}_{\mathcal{N}})\big) + T^{ref}{}_{\mathcal{N}} \tag{4.23}$$

By use of Equation (4.24)-(4.28) all the variables in the equation where made known to the system, and thus give the ability to calculate $H_{\mathcal{N}}$.

$$cp_{\mathcal{NS}} = cp_{\mathcal{S}} \overset{\mathcal{S}}{\star} P_{\mathcal{S},\mathcal{NS}} \tag{4.24}$$

$$Cp_{\mathcal{N}} = cp_{\mathcal{NS}} \overset{\mathcal{S} \in \mathcal{NS}}{\star} n_{\mathcal{NS}} \tag{4.25}$$

$$h^{ref}{}_{\mathcal{NS}} = h^{0}{}_{\mathcal{S}} \overset{\mathcal{S}}{\star} P_{\mathcal{S},\mathcal{NS}} \tag{4.26}$$

$$H^{ref}{}_{\mathcal{N}} = h^{ref}{}_{\mathcal{NS}} \overset{\mathcal{S} \in \mathcal{NS}}{\star} n_{\mathcal{NS}} \tag{4.27}$$

$$T^{ref}{}_{\mathcal{N}} = (e_{\mathcal{N}})\,.\,(T^{298}) \tag{4.28}$$

Chapter 5

Discussion

5.1 Results from model example

The model example provided a step-by-step approach to how the class instantiates an actual model by combining knowledge from the ontology and the topology. The class generated a suitable equation set with the required equations to constitute a mathematical model with zero degrees of freedom. The separation of nodes and arcs was done accordingly to the specification in each case topology and initialized the case using the various initialization schemes, with the following simulations performing in line with the expectations for a heat exchanger model. By giving the option of a faster initialization, the class helps the modeler with the main objective for this thesis, namely saving time.

5.2 Separation of nodes and arcs

The different nodes and arcs get separated into three categories for node and three for arcs. The nodes get separated into reservoirs, singles, and groups. The single and reservoir are stored in an identical structure only in different dictionaries. The idea, when doing this split, to build in some extra safety mechanisms to ensure that the reservoirs were properly initialized. The system-wide initialization function took care of that for reservoirs, but also all the other pieces of the model. As of now, the most important function the separation provides is helping the software distinguish between the different types of nodes when printing out the different options in the roll-down window. They could be combined into one dictionary to minimize the number of storage options.

The different categories for arcs are single, groups, and groups that connect group nodes. The single arcs and ”normal” groups work as they should, but a decision has to made about the last group. As there is a possibility that several layers of groups will arcs that connect group nodes, that are children of higher level group nodes, both connect the higher level nodes, the lower level nodes, and the higher level with the lower level nodes. An arc can then exist in several different groups. The easiest way of dealing with this is by saying all the are belong to the top level nodes. This solves the immediate problem, but can potentially create a new problem where the modeler does not want the same properties for all the arcs but want them divided into the groups they consist of on a lower level. If this is the case, the modeler would then waste time initializing all the arcs in question individually. For many systems, this type of group might even not be necessary, and the activation of the groups could be a choice the modeler has to make. The groups become instrumental when dealing with models such as the heat exchanger discussed in this thesis. When a model has two systems, that ideally would be modeled as distributed systems but are modeled as a series of lumped systems, connected, the grouping is beneficial to the user. As of now, they are connected to the top level group node.

5.3 Equation building

By using the selected states as starting points, the program builds a set of equations with zero degrees of freedom as the state ultimately on will depend on itself, another state, and a number of constants with values set by the user. To update the state, the equation calculating the state is always an integrator integrating the time-derivative of the state. Each ontology that is created has a finite set of variables at its disposal, of which a few are states. The modeler selects which states that are going to be calculated, and potentially a state will be dependent on another state. Such as mass and enthalpy, as shown in the heat exchanger example that is discussed earlier in this thesis. If the modeler only selects mass as the state, then the mass is still dependent on the enthalpy. Our solution to this problem is to treat the enthalpy as a constant and assume that the modeler wants to see how the system develops without any heat transport other than what is transported with the mass.

If a variable has more than one possible equation to calculate itself, the modeler has to choose which equation that is going to be used in the model. Since only one equation can be used, the equation will potentially not be valid for all the nodes as they can have different phases. A method to utilize the correct equation for the phase is, therefore, something to be worked on in the future. A potential method is to map the node and arc structures into intermediate structures for each phase. Each of the intermediates will use equations best suited to their phase, before being mapped back into one single structure before the integrator.

5.4 Initial state calculation

To check if the program can build an equation set to calculate the initial states, the modeler needs to propose a set of variables that possibly will be able to do so. As of now, the modeler does so without knowing anything about the structure in the variable tree. A future update to the software will be to show the structured tree in a GUI. Hence, the modeler will have the opportunity to locate potential equations and propose a specific set of variable needed for that equation. For simple mathematical models, with a low number of states, this can be a trivial task, but with increased model complexity locating and assessing the variables can be a difficult task. To help with the decision process, the program could do the locating scheme and variable checking that it does in this version, but in addition, it could also present the user with options for variables that can be set to complete the variable set. The question then is; what limitations should be set for the variables that can be suggested to the modeler? If the all possible combinations of variables are presented, the sheer number of possibilities could be confusing to the modeler unless they were properly categorized from ”easiest” to ”hardest” to calculate, though such a general categorization do not exist. Some restrictions could be, only giving variables directly involved in the equation where the state is a variable, limiting it to a set number of levels up and down in the variable tree, or only present variables that are secondary states. Making such limitation would decrease the number of possibilities significantly, but could also rule out potential options that are ”easy” on a higher or lower level.

5.5 Initialization

When working with large groups of nodes, often one or more nodes have properties different from the majority of the nodes, as they are added to the group due to their position in the user interface. A function to assign values to variables and states for such nodes, without interfering with the rest of the nodes in the group, was implemented. When such a node is initialized, the program does not allow the user to change the values of the node unless the same ”single function” is called with the node as the argument. This does not cause a problem as long as the modeler is thorough in the initialization process. If the modeler should happen to set the variables a random node in a group to values that will make the simulation crash, the error could be difficult to find in a large system of nodes with multiple groups at potentially many initialization levels. As of now, the system does not give any warning if an entity has been initialized on a lower level than the current initialization, but this can be a possible solution. Another solution might also be to give the user the option to overwrite the lower level initialization. This keeps the modeler constantly updated on what nodes have been initialized before, but also corrupt the idea of faster initialization a bit. In a large system, just giving the modeler a node number or name might not be enough information for the modeler to know if the node needs to be overwritten or not. Hence, the modeler needs to spend time investigating.

Chapter 6

Conclusion

This thesis has presented the implementation of an instating scheme for ontology-based model software. The class was implemented in a software called the *ProcessModellerSuite*, to reduce time spent on building the model and initializing it. The central concept used to reach this objective was the grouping of nodes and arc in the model topology, assuming uniform properties for all entities in a group. The implementation procedure, along with a model example as proof of concept, was presented. The model example combined the ontology and topology of a counter-current heat exchanger to instantiate a mathematical model with zero degrees of freedom. The different initialization schemes were tested by initializing the model on different levels, both for single entities and groups. A scheme to assess if the initial states could be calculated from a set of secondary states was added to the class, but not implemented in the model software. An example of the scheme's procedure was provided in the model example. The complete initialization was stored in the model software, with the ability to being loaded into a model utilizing the same structure and ontology, to allow reusing a previous initialization. The initialized variables and constants were used to simulate the model, showing a working implementation of the class in the *ProcessModellerSuite*.

Bibliography

Aris, R., 1978. Mathematical modelling techniques. Pitman, London.

Atkins, P., de Paula, J., 2006. Physical chemistry. Oxford.

Bieszczad, J., 2000. A framework for the language and logic of computer-aided phenomena-based process modeling. PhD thesis.

Bogusch, R., Lohmann, B., Marquardt, W., 2001. Computer-aided process modeling with modkit. Computers and Chemical Engineering (25), 963–995.
URL https://www.sciencedirect.com/science/article/pii/S0098135401006263

Butcher, J. C., 1996. A history of runge-kutta methods. Applied Numerical Mathematics (20), 247–260.
URL https://www.sciencedirect.com/science/article/pii/0168927495001085

Callen, H. B., 1985. Thermodynamics and an Introduction to Thermostatistics. John Wiley & Sons.

Decker, G., Mendling, J., 2009. Process instantiation. Data & Knowledge Engineering (68), 777–792.
URL https://www.sciencedirect.com/science/article/pii/S0169023X09000329

Elve, A. T., 2015. Ontology Design for Representation of mathematical Models. Master thesis, NTNU.

Elve, A. T., Preisig, H. A., 2019. From ontology to executable program code. Computers & Chemical Engineering (122), 383–394.
URL https://www.sciencedirect.com/science/article/pii/S0098135418309311

Esche, E., Bublitz, S., Tolksdorf, G., Repke, J.-U., 2018. Automatic decomposition of nonlinear equation systems for improved initialization and solution of chemical engineering process models. Computer Aided Chemical Engineering 44, 1387 – 1392.
URL http://www.sciencedirect.com/science/article/pii/B9780444642417502263

Geankopolis, C. J., 2003. Transport Processes and Separation Process Principles. Pearson Education.

Hanna, O. T., 1988. New explicit and implicit "improved euler" methods for the integration of ordinary differential equations. Computers & Chemical Engineering (12), 1083–1086.
URL https://www.sciencedirect.com/science/article/pii/0098135488870303

Haug-Warberg, T., 2006. Den termodynamiske arbeidsboken. Kolofon Forlag AS.

Kahtri, C. G., Rao, C. R., 1968. Solutions to some functional equations and their application to characterization of probability distributions. Sankhya (30), 167–180.
URL https://www.jstor.org/stable/25049527?seq=6#metadata_info_tab_contents

Kuntsche, S., Barz, T. amd Kraus, R., Arellano-Garcia, H., Wozny, G., 2011. Mosaic a web-based modeling environment for code generation. Computers & Chemical Engineering (35), 2257–2273.
URL https://www.sciencedirect.com/science/article/pii/S0098135411001128

Marquardt, W., Morbach, J., Wiesner, A., Yang, A., 2010. OntoCAPE: A Re-Usable Ontology for Chemical Process Engineering. Springer-Verlag, Berlin Heidelberg.

Petri, C. A., 1966. Communication with Automata. Technical Report RADC-TR-65-377. NewYork: Griffiss Air Force Base.

Piela, P., Epperly, T., Westerberg, K., A., W., 1991. Ascend: an object-oriented computer environment for modeling and analysis: The modeling language. Computers & Chemical Engineering (15), 53–72.
URL `https://www.sciencedirect.com/science/article/pii/009813549187006U`

Preisig, H. A., 2010. Constructing and maintaining proper process models. Computers & Chemical Engineering (34), 1543–1555.
URL `https://www.sciencedirect.com/science/article/pii/S0098135410000669`

Preisig, H. A., 2016. The ABC of modelling, Lecture notes TKP4106 & TKP 4135. NTNU.

Preisig, H. A., Elve, A. T., 2016. Ontology construction for multi-network models. Computer Aided Chemical Engineering (38), 1087–1092.
URL `https://www.sciencedirect.com/science/article/pii/B9780444634283501867`

Stephanopoulos, G., Henning, G., Leone, H., 1990. Model.la a modeling language fro process engineering–i. the formal framework. Computers & Chemical Engineering (14), 813–846.
URL `https://www.sciencedirect.com/science/article/pii/009813549087040V`

Westerweele, M. R., 2003. Five Steps for Building Consistent Dynamic Process Models and Their Implementation in the Computer Tool Modeller. Technische Universiteit Eindhoven.
URL `http://books.google.no/books?id=T3T8NwAACAAJ`

Westerweele, M. R., Laurens, J., 2008. Mobatec modeller - a flexible and transparent tool for building dynamic process models. Computers Aided Chemical Engineering (25), 1045–1050.
URL `https://www.sciencedirect.com/science/article/pii/S1570794608801800`

Yang, A., Marquardt, W., 2004. An ontology-based approach to conceptual process modelling. Computer aided Chemical Engineering (18), 1159–1164.
URL `https://www.sciencedirect.com/science/article/pii/S1570794604802591`

Appendix

Main template

```
# Automatically generated, do not edit!

"""
What:    Python simulation
Author:  ingolf
Contact: arne.t.elve(at)ntnu.no
Date:    2019-06-14 11:12:51
Model:   cc_HEX_single_nodes
Case:    done
"""

# Import packages:
import numpy as np   # Numerical python library
from scipy.integrate import ode   # Integrator in scipy
import matplotlib.pyplot as plt   # Data illustration
from funcUtils import IndexSet, khatriRao, blockReduce,
                               blockProduct    # Custom

from constants import *    # Import all constants
from networks import *    # Import all network variables
from selections import *
# from selections_ import *  # Import equation selections
# from initial_states_ import *
# ======== BODY =========#

# INDEX SETS:
N = IndexSet('node', mapping = [1, 2, 3, 4, 5, 6, 7, 8, 9,
                               10],
             blocking = [1, 1, 1, 1, 1, 1, 1, 1, 1, 1])
A = IndexSet('arc', mapping = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9
                               , 10],
             blocking = [1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1])
```

```
S = IndexSet('species', mapping = [0, 1, 2], blocking = [1,
                              1, 1])
N_x_S = IndexSet('node & species', mapping = [1, 2, 3, 4, 5
                             , 6, 7, 8, 9, 10],
             blocking = [1, 1, 1, 1, 1, 2, 2, 2, 2, 2])
A_x_S = IndexSet('arc & species', mapping = [0, 1, 2, 3, 4,
                              5, 6, 7, 8, 9, 10],
             blocking = [1, 1, 1, 1, 2, 2, 2, 2, 0, 0, 0])
R = IndexSet('species_conversion', mapping = [0], blocking
                             = [1])
N_x_R = IndexSet('node & species_conversion', mapping = [1,
                              2, 3, 4, 5, 6, 7, 8, 9, 10],
              blocking = [0, 0, 0, 0, 0, 0, 0, 0, 0, 0])

# Constant equations:
T_ref = np.multiply(en, T_298)
h_ref = np.transpose(np.dot(np.transpose(h_0),
                             P_node_species))
cp = np.transpose(np.dot(np.transpose(cpn), P_node_species)
                             )
Pnsas = np.dot(np.transpose(P_node_species), P_arc_species)
Fn = khatriRao(F_mass_volumetric, [N, A], Pnsas, [N_x_S,
                             A_x_S])



# INTEGRATING FUNCTION:
def derivative(t, n, H):
  """
  t: time
  state: n, H
  Integrating function:
  dxdt = derivative(t, state)

  Note: sequence of variables depends on integrator. Using
                              scipy's ode requires
  time before state, while odeint needs state before time.
                              Also the integrator
  want  flat vectors,  no column vectors.  I, therefore,
                              transpose  the vectors
  manually. Normal transpose is not sufficient.
  """
  # EQUATIONS:
  H_ref = blockReduce(h_ref, S, N_x_S, n)
  Cp = blockReduce(cp, S, N_x_S, n)
  T = np.add((np.multiply((np.reciprocal(Cp)), (np.subtract
                              (H, H_ref)))), T_ref)
  qhat = np.multiply((np.multiply((-(U)), A_heat)), (np.dot
                              (np.transpose(F_energy_heat
                              ), T)))
  ntot = blockReduce(ens, S, N_x_S, n)
  p = np.multiply((np.multiply((np.multiply((np.reciprocal(
                              Vg)), ntot)), T)), Rg)
  dir = np.sign(np.dot(np.transpose(F_mass_volumetric), p))
  s_flow = np.multiply(half, (np.add((np.abs(
                              F_mass_volumetric )), (np.
                              multiply(F_mass_volumetric,
                               np.transpose(dir))))))
  Vhat = np.multiply((np.multiply((-(kappa)), A_cross)), (
                              np.dot(np.transpose(
                              F_mass_volumetric), p)))
  Hhat = np.multiply(Vhat, (np.dot(np.transpose(s_flow), (
                              np.multiply((np.reciprocal(
                              Vg)), H)))))
  Hdot = np.add((np.dot(F_mass_volumetric, Hhat)), (np.dot(
                              F_energy_heat, qhat)))
  c = khatriRao(np.reciprocal(Vg), [N], n, [N_x_S])
  chat = np.dot(np.transpose((khatriRao(s_flow, [N, A],
                              Pnsas, [N_x_S, A_x_S]))), c
                              )
  nhat = khatriRao(Vhat, [A], chat, [A_x_S])
  ndot = np.dot(Fn, nhat)
  return np.multiply(Selection_ndot, ndot), np.multiply(
                              Selection_Hdot, Hdot)

def integrand(t, state):
  # HACK: Integrator give flat state vectors while our
                              model is column vector
  n = state[0:15, np.newaxis]
  H = state[15:25, np.newaxis]
  ndot, Hdot = derivative(t, n,  H)
  return np.concatenate([np.transpose(ndot)[0], np.
                              transpose(Hdot)[0]])

# INTEGRATOR
data = {}
data['n'] = []
data['H'] = []
data['t'] = []
dt = 0.1
```

```
t_start = 0
t_end   = 10.
                                              # Handcoded!!!
integrator = ode(integrand).set_integrator('dop853')
state = np.concatenate([np.transpose(n)[0], np.transpose(H)
                                              [0]])
integrator.set_initial_value(state, t_start)
while integrator.successful() and integrator.t < t_end :
  state = integrator.integrate(integrator.t + dt)
  data['t'].append(integrator.t)
  data['n'].append(state[0:15, np.newaxis])
  data['H'].append(state[15:25, np.newaxis])


data['n'] = np.transpose(data['n'])[0]
for i, (dat) in enumerate(data['n']):
  plt.plot(data['t'], dat)
plt.legend()
plt.show()

data['H'] = np.transpose(data['H'])[0]
for i, (dat) in enumerate(data['H']):
  plt.plot(data['t'], dat)
plt.legend()
plt.show()
```

Information files

```
# Automatically generated, do not edit!

"""
What:     Python initialization file
Author:   ingolf
Contact:  arne.t.elve(at)ntnu.no
Date:     2019-06-13 16:43:32
Model:    cc_HEX_single_nodes
Case:     done
"""

# Import packages:
import numpy as np                                      # Num
                                          erical python library

# Imutable variables:S
```

```
en = np.array([[1.0], [1.0], [1.0], [1.0], [1.0], [1.0], [1
                              .0], [1.0], [1.0], [1.0]])
ens = np.array([[1.0], [1.0], [1.0], [1.0], [1.0], [1.0], [
                              1.0], [1.0], [1.0], [1.0], [1
                              .0], [1.0], [1.0], [1.0], [1.
                              0]])
h_0 = np.array([[1.0], [1.0], [1.0]])
n = np.array([[15500.0], [13500.0], [12000.], [11000.], [
                              10000.0], [7500.0], [7500.0],
                               [6000.0], [6000.0], [3000.0]
                              , [3000.0], [2000.0], [2000.0
                              ], [1000.0], [1000.0]])
H = np.array([[150000000.0], [50000000.0], [50000000.0], [
                              50000000.0], [50000000.0], [1
                              .0], [1.0], [1.0], [1.0], [1.
                              0]])
U = np.array([[0.0], [0.0], [0.0], [0.0], [0.0], [0.0], [0.
                              0], [0.0], [10000.0], [10000.
                              0], [10000.0]])
cpn = np.array([[75.0], [75.0], [75.0]])
half = np.array(0.5)
kappa = np.array([[0.000001], [0.000001], [0.000001], [0.
                              000001], [0.000001], [0.
                              000001], [0.000001], [0.
                              000001], [0.000001], [0.
                              000001], [0.000001]])
Vg = np.array([[1.0], [1.0], [1.0], [1.0], [1.0], [1.0], [1
                              .0], [1.0], [1.0], [1.0]])
T_298 = np.array(298.0)
A_cross = np.array([[0.1], [0.1], [0.1], [0.1], [0.1], [0.1
                              ], [0.1], [0.1], [10.], [10.]
                              , [10.]])
Rg = np.array(8.314)
```

```
# Automatically generated, do not edit!

"""
What:    Python initialization file
Author:  ingolf
Contact: arne.t.elve(at)ntnu.no
Date:    2019-06-13 16:43:32
Model:   cc_HEX_single_nodes
Case:    done
"""
```

```
# Import packages:
import numpy as np                                              #
                                        Numerical python library

# Imutable variables:
F_energy_heat = np.array(
[[ 0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.],
 [ 0.,  0.,  0.,  0.,  0.,  0.,  0.,  0., -1.,  0.,  0.],
 [ 0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0., -1.,  0.],
 [ 0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0., -1.],
 [ 0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.],
 [ 0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.],
 [ 0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  1.],
 [ 0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  1.,  0.],
 [ 0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  1.,  0.,  0.],
 [ 0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.]])
F_mass_volumetric = np.array(
[[-1.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.],
 [ 1., -1.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.],
 [ 0.,  1., -1.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.],
 [ 0.,  0.,  1., -1.,  0.,  0.,  0.,  0.,  0.,  0.,  0.],
 [ 0.,  0.,  0.,  1.,  0.,  0.,  0.,  0.,  0.,  0.,  0.],
 [ 0.,  0.,  0.,  0., -1.,  0.,  0.,  0.,  0.,  0.,  0.],
 [ 0.,  0.,  0.,  0.,  1., -1.,  0.,  0.,  0.,  0.,  0.],
 [ 0.,  0.,  0.,  0.,  0.,  1., -1.,  0.,  0.,  0.,  0.],
 [ 0.,  0.,  0.,  0.,  0.,  0.,  1., -1.,  0.,  0.,  0.],
 [ 0.,  0.,  0.,  0.,  0.,  0.,  0.,  1.,  0.,  0.,  0.]])
F_species_volumetric = np.array(
[[-1.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.],
 [ 1., -1.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.],
 [ 0.,  1., -1.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.],
 [ 0.,  0.,  1., -1.,  0.,  0.,  0.,  0.,  0.,  0.,  0.],
 [ 0.,  0.,  0.,  1.,  0.,  0.,  0.,  0.,  0.,  0.,  0.],
 [ 0.,  0.,  0.,  0., -1.,  0.,  0.,  0.,  0.,  0.,  0.],
 [ 0.,  0.,  0.,  0.,  1., -1.,  0.,  0.,  0.,  0.,  0.],
 [ 0.,  0.,  0.,  0.,  0.,  1., -1.,  0.,  0.,  0.,  0.],
 [ 0.,  0.,  0.,  0.,  0.,  0.,  1., -1.,  0.,  0.,  0.],
 [ 0.,  0.,  0.,  0.,  0.,  0.,  0.,  1.,  0.,  0.,  0.]])
P_arc_species = np.array(
[[ 1.,  1.,  1.,  1.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0
                                         .,  0.,  0., 0.],
 [ 0.,  0.,  0.,  0.,  1.,  0.,  1.,  0.,  1.,  0.,  1.,  0
                                          .,  0.,  0., 0.],
 [ 0.,  0.,  0.,  0.,  0.,  1.,  0.,  1.,  0.,  1.,  0.,  1
                                          .,  0.,  0., 0.]])
```

```
P_conversion_species = np.array(
[])
P_node_species = np.array(
[[ 1.,  1.,  1.,  1.,  1.,  0.,  0.,  0.,  0.,  0.,  0.,  0
                                  .,  0.,  0., 0.],
 [ 0.,  0.,  0.,  0.,  0.,  1.,  0.,  1.,  0.,  1.,  0.,  1
                                   .,  0.,  1., 0.],
 [ 0.,  0.,  0.,  0.,  0.,  0.,  1.,  0.,  1.,  0.,  1.,  0
                                   .,  1.,  0., 1.]])
```