**Ontology-driven automation of process modelling and simulation**

Alberto Rodríguez-Fernández, Vinay Kumar Gautam, Heinz A Preisig\*

*Department of Chemical Engineering, Norwegian University of Science and Technology, N-7491 Trondheim, Norway*

*Heinz.Preisig@chemeng.ntnu.no*

**Abstract**

NTNU’s **Pro**cess **Mo**delling suite, ProMo, adopts a rigorous ontology-driven modelling paradigm that aligns with the foundational tenets of physical, chemical, and biological systems. This approach serves as the basis for the systematic development of models. Creating a coherent variable/expression list, manifested as a multi-bipartite graph, forms the bedrock for assembling fundamental building blocks. These building blocks, in turn, expedite the construction of complex models, ensuring coherence across the modelling lifecycle. This work showcases our method for preparing model data to be integrated into the code generation process.

**Keywords**: Ontology, simulation, computational engineering, automatic code generation.

* 1. **Ontology-Driven Simulations**

A user-friendly graphical interface augments the user experience when utilising simulation software. Block diagrams are the most common technology for visualisation. They connect activities, often spiced with logical components, implementing a logic-driven control structure. Some application domains, like process control, use them as a preferred tool, and chemical engineering uses flowsheeting as the primary plant design tool. People grab pictorial information much quicker than corresponding textual. It is not a surprise then that these tools were some of the very early developments when computing arrived, and various tools emerged consequently. Monsanto's FLOWTRAN was released in 1966, triggering several projects, such as ASPEN (1981), HYSYS (ASPEN), UniSim (Honeywell). The need for dynamic simulators started SPEEDUP (Imperial College) and later gPROMS (Imperial College, PSE, now Siemens). The last 20 years saw a development towards an ontology-driven model-development environment with ASCENT (Carnegie Mellon), MODEL.A (MIT), MODKIT (RWTH), Modeller (TU/E), and ProMo (NTNU), driven by the objective to produce internally correctly structured models. NTNU's ProMo builds on the concept of fundamental entities, representing the modelled process as a communicating network of these entities. NTNU has been working with a graphical representation for teaching and designing processes, with the graphical language eventually evolving into a standard (Preisig, 2022).

ProMo constructs first an ontology that captures the main structures and items. In the second step, the ontology is used to build a *variable ∶= expression* bipartite graph, which in the third step are the ingredients for defining the behaviour of the entities' specialised varieties by a modelling specialist. The next step is the construction of process models, done by what we now refer to as a translator, a person who learns about the process and maps it into a process model using the expert's fundamental building blocks. The final stage is the generation of target code, which is a parameterised simulation in either a stand-alone task or a script for the target simulation environment. In contrast to most other simulation systems where the equations are spread over different model modules, ProMo implements a **single module** with all the equations.

When users interact with the different modules, they provide information in several ways, such as variables and defining expressions, construction of entity models, visual construction of the process model, and instantiation of variables. The Task Factory module of ProMo uses all this information to generate the target code automatically. A templating engine performs the translation to a specific language as the last step in the code generation. This approach allows for great flexibility, making it possible to switch between different languages, such as C++, Python, or Matlab, by selecting the appropriate template. Modifying the templates so the generated code can be used as a standalone file or as input for another program like an orchestrator is possible. This paper focuses on transforming the user-supplied information into a form that the templating engine can consume. In Section 2, we introduce the structure of the data entered by the user, followed by a description of the input that the templating engine requires. Section 3 details the data processing necessary to fulfil those requirements. Section 4 summarises the procedure and presents other uses it can have in the future.

* 1. **The ProMo Automation Workflow**
     1. *User input data*
        1. *Entity Models*

Entity models represent elementary parts of the modelled system. The CWA (Preisig, 2022) defines a set of elementary entities. Their definition in the context of the modelled process depends on the granularity required to capture the modelled object’s behaviour. It is thus application dependent.  Each entity model contains a subset of variables and equations from the bipartite graph, forming a coupled equation forest. Mathematically, each entity model is seen as an input/output function, and the equation forest computes variables as a function of other variables. On the function level, the variables are classified as follows:

* *Input* variables originate from other entities and are the external quantities needed to calculate the *output* variables.
* *Output* variables are the result of computing the entity models. They are the inputs to other entity models.
* *Instantiation* variables represent quantities related to the characteristics of the part of the modelled system the entity model describes. Initial conditions and parameters are typical class members, and the user must introduce numerical values later in the modelling process.
* The variable/equation forest links all the variables in an entity by defining equations. Each *output* variable is the root of one tree in the forest. Each node in a tree will contain either a variable or an equation. Variable nodes can have at most one child, its defining equation. Equation nodes can have one child for each variable on the right-hand side of the equation. This structure ensures that going from the leaves of a particular tree to the root yields the sequence of equations necessary to compute the corresponding *output* variable. Finally, variables in each tree are available to all the other trees, ensuring the uniqueness of the nodes in the forest.

In addition, each entity model contains a unique index set name. An entity model is complete when all the variables are *input*, *instantiation* or have a defining equation in the variable/equation forest. This completeness guarantees the feasibility of computing all variables if the entity model has access to the *input* variables and numerical values for all *instantiation* variables. Only complete entity models can be used as blocks to construct the topology of the process model.

* + - 1. *Topology graph*

The Modeller module of ProMo allows a user to map a process into a process model using a graphical tool. This visual representation is automatically converted and stored as a topology graph with nodes that store “tokens” and arcs that control their transfer between nodes (Preisig 2021). The nodes and arcs contain references to the entity models. That is why, while constructing this graph, we simultaneously define the connections between entity models. As a rule, entity models can only receive *input* variables from others directly connected to them.

The topology graph presents a visual and intuitive way of splitting a system into smaller components and their relationships. The entity models help to distinguish between these components and provide the necessary mathematical structure. The graphic nature of this representation also comes in handy to assign numerical values to the *instantiation* variables, allowing the user to orient himself in a commonly complex system.

* + - 1. *Instantiation*

All *instantiation* variables in an entity model require a numerical value. The user provides these values, transforming a general representation of the model into a parameterised instance. ProMo stores this information as a group of equations (assignments) in the form:

|  |  |
| --- | --- |
|  | (1) |

where x is an *instantiation* variable, and # is a numerical value.

A more advanced case appears when the user assigns numerical values to variables not classified as *instantiation*. This action triggers a restructuring of the variable/equation forest of one or multiple entity models. It is beyond the scope of this paper to go into the details of the required manipulations, but the resulting entity models behave in the same way as the previously defined ones.

Several checks ensure that no *instantiation* variable remains without a numerical value. Failure to comply with this condition will lead to a mathematical representation of the model that is not solvable.

* + 1. *Processed output data*

The final step in the ProMo is the automatic code generation. The task factory module writes all the necessary information into a format defined by the target language. A templating engine takes the essential data and uses language-specific templates to complete this step. The required input data to the templating engine consists of:

1. A list of variables used and values for the ones marked as *instantiation*.
2. A list of equations used and their computing order.
3. Index sets indicate the relation of entity models to elements in the topology.
4. Information about language-specific ways of representing the variables and equations.

Items 1 and 2 require some additional processing of the user input data, which will be detailed in the following section. The data in items 3 and 4 is already available at this stage. The former comes from the topology graph. Each index set name belongs to one of the entity models used, and all nodes and arcs referencing the entity model belong to the set. The latter originates from creating the bipartite graph in the first steps of our modelling journey. The user can supply additional data to document the variables and equations at that step. This data can be marked as language-specific to comply with language rules or guidelines specified by different communities, ensuring the resulting code aligns with the preferences and needs of its end-users.

ProMo generates the code automatically using a templating engine. The general user is advised not to change the automatically generated code manually. Instead, to go back and make necessary changes to previous steps.

* 1. **Equation sequencing**

As previously mentioned, our starting point is the topology graph, where each node and arc are linked to an entity model and has a set of already instantiated variables (Figure 1a). We use different incidence matrices to capture the relationship between nodes and arcs under certain constraints (nodes and arcs containing specific tokens, transfer mechanisms, etc.). These matrices are variables in our bipartite graph and will preserve the information of the topology graph once we move on from this representation.

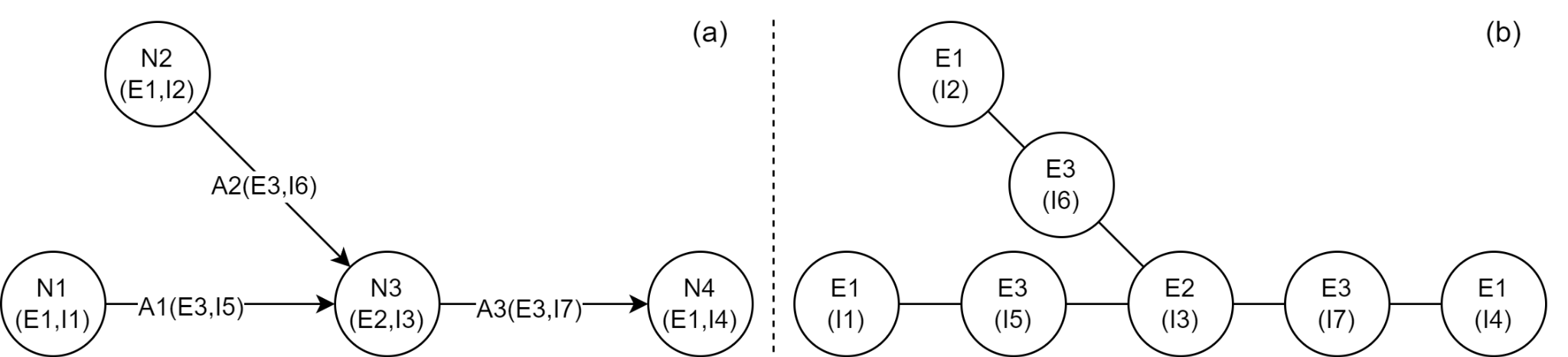


Figure 1. (a) Example topology graph: each node and arc have a link to an entity and a group of instantiated variables. (b) Entity model graph constructed from the topology graph on the left.

Our first step is constructing an undirected graph where each vertex corresponds to an entity model and associating it with the corresponding values of the *instantiation* variables. We add one vertex for each node and arc in the topology graph. At this stage, the entity models are the main focus, and it is irrelevant whether there was a link to one type of element or the other in the previous graph. Edges between the new graph's vertices indicate an arc going in or out from a node in the topology graph. Figure 1b shows the resulting graph obtained from the one shown in Figure 1a. In this new graph, adjacency between two vertices means that the corresponding entities can exchange *input* and *output* variables, as previously mentioned, which simplifies the process of finding the required *input* variables.

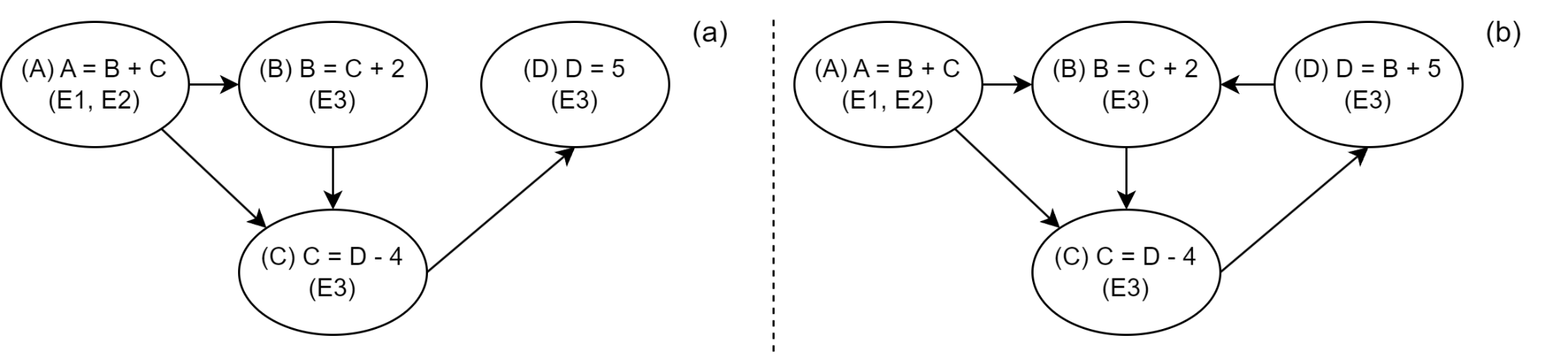
At this point, variables and equations are spread over the entity model graph, each vertex containing its own subset. Next, we will create a new directed graph containing only variables and equations. We will use this final form of our data to find the correct computing order for the equations.

As illustrated in Figure 2a, each vertex in the new graph contains a tuple of three elements: a variable, an equation, and a list of index set names corresponding to the entity models where this specific variable/equation combination appears. Arcs between vertices imply that the equation in the tail vertex contains the variable in the head one.

At first, we add variable/equation pairs representing balance or conservation for the state variables. If the same variable/equation combination appears in multiple entity models, we group all the index set names together in one vertex instead of creating a new one each time. This rule is observed in general and implies that each combination appears only once in the graph. Each equation provides a new group of variables that are added to a queue.

From now on, we process the variables in the queue. First, we search to find their defining equations. In the case of *input* variables, this implies a search in the adjacent entity models. For *instantiation* variables, an equation like Eq. (1) is constructed for each instantiated value. Given that all utilised entity models adhere to the completeness condition outlined in Subsection 1.1.1, each of the remaining variables will possess a defining equation within the variable/equation forest.  For each new variable/equation pair, a new vertex is created. If the pair exists already in the graph, the index set name is added to the existing vertex. In both cases, we add an arc connecting the vertex containing the parent equation to the vertex with the children variable/equation pair. New variables are added to the queue when a vertex with a new equation is created. This process is repeated until the queue is empty. Figure 2a shows an illustrative example of this type of graph.

At last, we are ready to find the computation order for the equations. We have transformed our data and ended up with what is, at first sight, a dependency resolution or scheduling problem (Beeri et al. 1981). Our solution uses a modified version of the topological sort algorithm (Kahn 1962), tailored to accommodate the following two constraints in our computational model.

Figure 2. Example variable/equation graph. (a) Simple case. (b) Case with a cycle.

First, equations corresponding to the *instantiation* variables should be collected independently. Usually, this type of equation is evaluated only once at the beginning of the generated code, so it makes sense to separate them. From looking at Eq. (1), it is evident that vertices containing this type are always head vertices. This understanding facilitates their removal from the graph without affecting the ordering of the remaining vertices.

Second, the variable/equation digraph is not restricted to being a Directed Acyclic Graph (DAG) (See Figure 2b), which implies that the topological sorting algorithm will fail if cycles are not handled appropriately. Cycles in this graph indicate the presence of equations that need to be solved simultaneously. They commonly appear as part of strongly connected components (SCCs) (Tarjan 1972). All the equations in vertices belonging to an SCC must be solved at the same time. In our case, we use Tarjan’s algorithm (Tarjan 1972) with Nuutila’s modifications (Nuutila 1994) to find all SCCs and replace each with a single vertex. Each of these vertices contains all the equations in the corresponding SCC and is labeled accordingly so the templating engine can process them correctly.

The subsequent procedure is straightforward. We execute the topological sorting algorithm and reverse the resulting order to obtain the computation order for the equations. A list of index set names is attached to each equation so the correct values for the variables can be used.

After all these steps, the data is ready to be consumed by the templating engine. The resulting code contains all the necessary equations in one place, disregarding the entity models where they were used. This is only possible because the bipartite variable/equation graph is built as a whole at the beginning of the process and referenced from there on. This approach reduces the number of mistakes derived from defining the same equations in several places. Furthermore, it facilitates the analysis for consistency of the equations used in the model.

* 1. **Conclusions**

To automatically generate code using the information provided by users at different levels, ProMo uses a templating engine with templates written for each target language. Several data processing steps are necessary to convert the user's input to the information that feeds the template manager. First, we transform the topology graph into one where the entity models are the main component, which allows for an effective search of equations corresponding to *input* variables in the entity models. Next, we collect all variables and equations into a directed graph, allowing the use of topological sorting to find the computing order for the equations. Equations for instantiation variables and systems of equations are handled specially and labelled accordingly. This procedure has been implemented in the Task Factory module of ProMo, and the current tests have shown great success.

**Acknowledgements**

VIPCOAT H2020-NMBP-TO-IND-2020, Grant Agreement No. 952903; MEDIATE-net project, DigiPass CSA, HORIZON-CL4-2023-RESILIENCE- 01-39, Grant Agreement No. 101138510, NanoLodge NFR 299363, MEDIATE a European net project; Bio4Fuels (Norwegian Centre for Environment-friendly Energy Research (FME), project 257622).

**References**

A.B. Kahn, 1962, Topological Sorting of Large Networks, Commun. ACM, 5(11), 558–562.

C. Beeri and M.Y. Vardi, 1981. The implication problem for data dependencies. In: Even, S.,

H.A. Preisig, 2010, Constructing and maintaining proper process models, Comp & Chem Eng, 34(9), 1543-1555.

H.A. Preisig, 2021, Ontology-Based Process Modelling-with Examples of Physical Topologies, Processes, 9.

H.A. Preisig et al., 2022, ModGra - a Graphical representation of physical process models; CWA 1796, CEN-GENELEC Management Centre, Brussels.

E. Nuutila. and E. Soisalon-Soinen, 1994, On finding the strongly connected components in a directed graph, Information Processing Letters, 49(1), 9-14.

O. Kariv, (eds) Automata, Languages and Programming. ICALP 1981. Lecture Notes in Computer Science, vol 115. Springer, Berlin, Heidelberg.

R. Tarjan, 1972, Depth-first search and linear graph algorithms, SIAM Journal of Computing, 1(2):146-160.