From Petri Nets to Guard-Stage-Milestone Models

Viara Popova and Marlon Dumas

Institute of Computer Science,
University of Tartu, J. Liivi 2,
Tartu 50409, Estonia
{viara.popova,marlon.dumas}@ut.ee

Abstract. Artifact-centric modeling is an approach for modeling business processes based on business artifacts, i.e., entities that are central for the company's operations. Existing process mining methods usually focus on traditional process-centric rather than artifact-centric models. Furthermore, currently no methods exist for discovering models in Guard-Stage-Milestone (GSM) notation from event logs. To bridge this gap, we propose a method for translating Petri Net models into GSM which gives the possibility to use the numerous existing algorithms for mining Petri Nets for discovering the life cycles of single artifacts and then generating GSM models.

Keywords: Artifact-Centric Modeling, Guard-Stage-Milestone, Petri Nets, Process Mining.

1 Introduction

Artifact-centric modeling is a new promising approach for modeling business processes based on the so-called business artifacts [2,9] - key entities driving the company's operations and whose life cycles define the overall business process. An artifact type contains an information model with all data relevant for the entities of that type as well as a life cycle model which specifies how the entity can progress responding to events and undergoing transformations from its creation until it is archived.

Most existing work on business artifacts has focused on the use of life cycle models based on variants of finite state machines. Recently, a new approach was introduced - the Guard-Stage-Milestone (GSM) meta-model [5,6] for artifact life cycles which is more declarative than the finite state machine variants, and supports hierarchy and parallelism within a single artifact instance.

Some of the advantages of GSM [5,6] are in the intuitive nature of the used constructs which reflect the way stakeholders think about their business. Furthermore, its hierarchical structure allows for a high-level, abstract view on the operations while still being executable. It supports a wide range of process types,

from the highly prescriptive to the highly descriptive. It also provides a natural, modular structuring for specifying the overall behavior and constraints of a model of business operations in terms of ECA-like rules.

Currently, GSM models are created manually which can require a lot of effort and domain knowledge. The area of Process Mining focuses on developing methods for automatic discovery and analysis of process models such as conformance checking, repair and so on. Most existing methods consider only process-centric models, most often Petri Nets (PN) and no methods have been developed that are applicable to GSM models. In order to bridge the gap, this paper proposes a method for translating PN models to models in GSM. As a result, existing methods can be applied for discovering the life cycles of the separate artifacts which can then be represented as GSM models. Furthermore, manually created PN can be translated to GSM which allows to explore and reuse existing model libraries, case studies and domain knowledge.

The method presented in this paper is implemented as a software plug-in for ProM, a generic open-source framework and architecture for implementing process mining tools in a standard environment [13] which is the *de facto* industry standard in process mining. The implementation is part of the ArtifactModeling package which is available from www.processmining.org.

The paper is organized as follows. Section 2 introduces the case study used for illustration. Section 3 gives a brief overview of the modeling approaches needed for the presentation of the contribution of the paper. Sections 4, 5 and 6 introduce the method for translating PN models to GSM proposed in this paper. Finally, section 7 concludes the paper.

2   Case Study

As a case study we consider a model of an order-to-cash process as follows. The process starts when the manufacturer receives a purchase order from a customer for a product that needs to be manufactured. This product typically requires multiple components or materials which need to be sourced from suppliers. To keep track of this process, the manufacturer first creates the so-called work order which includes multiple line items - one for each required component. Multiple suppliers can supply the same materials thus the manufacturer needs to select suppliers first then place a number of material orders to the selected ones.

Suppliers can accept or reject the orders. If an order is rejected by the supplier then a new supplier is found for these components. If accepted, the order is assembled and delivered and, in parallel, an invoice is sent to the manufacturer. When all material orders for the same purchase order are received, the product is assembled and delivered to the customer and an invoice is sent for it.

The customer can cancel a purchase order. The request for cancellation is forwarded to the suppliers and assessed. If accepted, cancellation fee is determined, otherwise the order is delivered and invoiced in full.

Figure 1 shows one way of modeling the order-to-cash example using Proclet notation as will be described in the next section.

3   Background

We first give the necessary background in order to present the PN to GSM translation method by a very brief introduction to both modeling approaches.

Petri nets [8] are an established tool for modeling and analyzing workflow processes. They have been used in a wide variety of contexts and a great number of the developed process mining techniques assume or generate Petri Nets.

A PN is a directed bipartite graph with two types of nodes called *places* (represented by circles) and *transitions* (represented by rectangles) connected with arcs. Intuitively, the transitions correspond to activities while the places are conditions necessary for the activity to be executed. Transitions which correspond to business-relevant activities observable in the actual execution of the process will be called visible transitions, otherwise they are invisible transitions. A labeled PN is a net with a labeling function that assigns a label (name) for each place and transition. Invisible transitions are assigned a special label τ.

An arc can only connect a place to a transition or a transition to a place. A place p is called a pre-place of a transition t iff there exists a directed arc from p to t. A place p is called a post-place of transition t iff there exists a directed arc from t to p. Similarly we define a pre-transition and a post-transition to a place.

At any time a place contains zero of more tokens. The current state of the PN is the distribution of tokens over the places of the net. A transition t is enabled iff each pre-place p of t contains at least one token. An enabled transition may fire. If transition t fires, then t consumes one token from each pre-place p of t and produces one token in each post-place p of t.

In order to use the PN notation for modeling artifact-centric systems, we need a generalization of PN which reflects the artifact structure and interactions. For this, **Proclets** [12] can be used as discussed in the following paragraphs.

A *proclet* $P = (N, ports)$ is a labeled PN, which describes the internal life cycle of one artifact, and a set of ports, through which P can communicate with other proclets. Relations between several proclets are described in a *proclet system* $\mathcal{P} = (\{P_1, \ldots, P_n\}, C)$ consisting of a set of proclets $\{P_1, \ldots, P_n\}$ and a set C of *channels*. Each channel $(p, q) \in C$ connects two ports p and q of two proclets of $\mathcal{P}$ which send and receive messages along these channels. The channels also reflects the relations between entity types: annotations at the ports define how many instances of a proclet interact with how many instances of another proclet. Each half-round shape represents a port: the bow indicates the direction of communication. A dashed line between 2 ports denotes a channel of the system. Creation and termination of an artifact instance is expressed by a respective transition, drawn in bold lines.

Fig. 1 shows one way of modeling the order-to-cash example as a proclet system of two proclets that model artifacts PurchaseOrder and MaterialOrder.

Proclets are suitable for describing multi-artifact systems due to the annotations $1, ?, +$ in the ports [12]. The first annotation, called *cardinality*, specifies how many messages one proclet instance sends to (receives from) other instances when the attached transition occurs. The second annotation,
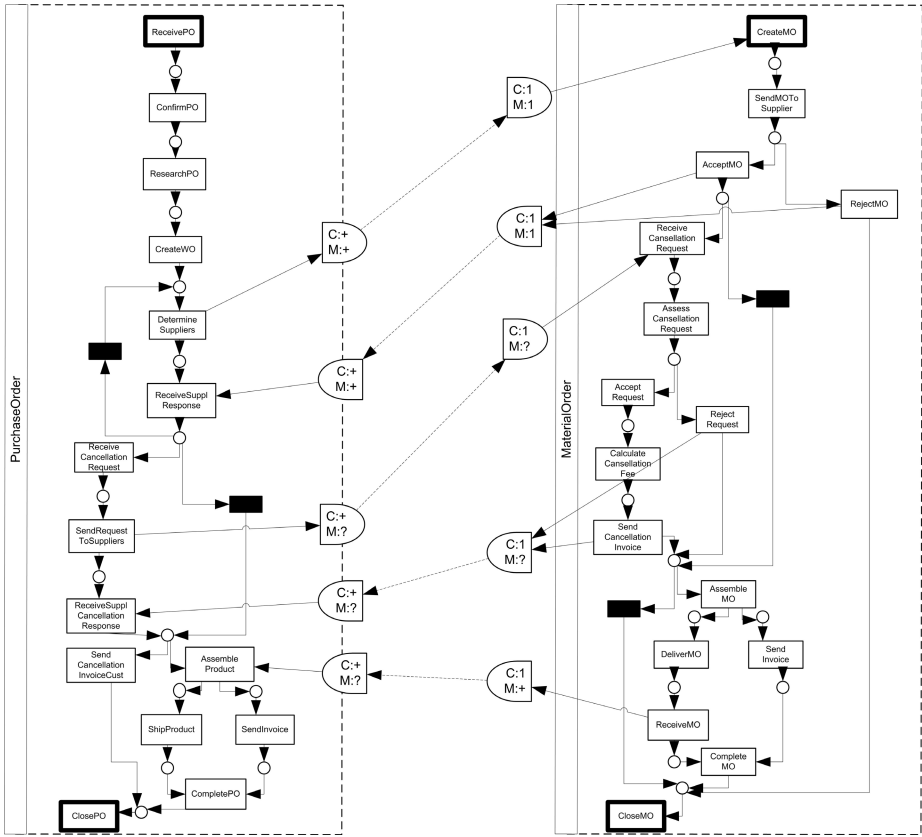
Fig. 1. The Proclet model of the Order-to-Cash example

multiplicity, specifies how often this port is used in the lifetime of a proclet instance. For example, the port of DetermineSuppliers has cardinality + and multiplicity + denoting that a PurchaseOrder instance sends out one or more messages with ordered items to multiple MaterialOrders and this can happen once or multiple times in the lifetime of the PurchaseOrder instance.

In this paper we concentrate on the artifact life cycle rather than the communication between artifacts. Therefore we only consider single proclets which are in fact Petri nets. In the rest of this paper we talk about translating Petri net models to GSM models. All the results are in fact applicable for proclets and thus for single artifact life cycle models.

The **Guard-Stage-Milestone meta-model** [5,6] provides a more declarative approach for modeling artifact life cycles which allows a natural way for representing hierarchy and parallelism within the same instance of an artifact and between instances of different artifacts.

The key GSM elements for representing the artifact life cycle are stages, guards and milestones which are defined as follows.

Milestones correspond to business-relevant operational objectives, and are achieved (and possibly invalidated) based on triggering events and/or conditions over the information models of active artifact instances. Stages correspond to clusters of activity preformed for, with or by an artifact instance intended to achieve one of the milestones belonging to the stage. Guards control when stages are activated, and, as with milestones, are based on triggering events and/or conditions. A stage can have one or more guards and one or more milestones. It becomes active (or open) when a guard becomes true and inactive (or closed) when a milestone becomes true.

Furthermore, sentries are used in guards and milestones, to control when stages open and when milestones are achieved or invalidated. Sentries represent the triggering event type and/or a condition of the guards and milestones. The events may be external or internal, and both the internal events and the conditions may refer to the artifact instance under consideration, and to other artifact instances in the artifact system.

4   Petri Nets to GSM Models - The General Approach

A straightforward approach to translating PNs to GSM models would proceed as follows. The visible transitions of the PN represent activities which are part of the business process. Therefore it is logical to represent them as atomic stages where the activity corresponds to the task associated with the stage. The control flow of the PN can then be encoded using the guards and milestones of these stages.

It is possible to use an explicit representation of the places of the PN using a collection of variables which will be part of the information model of the GSM component. These variables will be assigned true or false simulating the presence or absence of tokens in the places. This will be a relatively intuitive approach for designers skilled in the PN notation. However we argue that this would make the model less intuitive to the user and the relations between the tasks and stages become implicit and not easy to trace. Here we take a different approach which will be discussed in this section at a more general level and in the next sections in more detail.

The intuition behind this approach is that the immediate ordering relations between transitions in the PN are extracted, translated into conditions and combined using appropriate logical operators (for AND- and OR-splits and joins) into sentries which are then assigned to the guards. The milestones are assigned sentries that depend on the execution of the task associated with the stage - a milestone is achieved as soon as the task is executed and is invalidated when the stage is re-opened.

As an example, consider the transition ResearchPO from the order-to-cash model in Fig. 1. It can only be executed after the transition ConfirmPO has been executed and there is a token in the connecting place. This can be represented

as a part of a GSM model in the following way. Both transitions are represented by atomic stages. The guard of the stage ResearchPO has a sentry with expression (given here informally) "on ConfirmPOMilestone.achieved()" and will become true when the event of achieving the milestone of stage ConfirmPO occurs. The milestone of stage ResearchPO has a sentry "on ResearchPOTask.executed()" and will become true when the associated task is executed. Similarly the milestone of ConfirmPO has a sentry "on ConfirmPOTask.executed()".

While this example is very straightforward, a number of factors can complicate the sentries. Most importantly, we need to consider the possibility of revisiting a stage multiple times - this can be the case when the corresponding transition in the PN is part of a loop. Furthermore the transition might depend on the execution of multiple pre-transitions together and this cannot be represented using events - conditions need to be used instead. The conditions should express the fact that new executions of the pre-transitions have occurred. This means that the last execution of each relevant pre-transition occurred after the last execution of the transition in focus but also after every "alternative" transition, i.e., transition that is an alternative choice.

For example consider the transition CompletePO in Fig. 1 which can only fire if both ShipProduct and SendInvoice have fired. While this is not part of the model, imagine the hypothetical situation that CompletePO, ShipProduct and SendInvoice were part of a loop and could be executed multiple times. Since a sentry cannot contain multiple events, the guard of CompletePO has to be expressed by conditions instead. The naïve solution "if ShipProductTask.hasBeenExecuted() and SendInvoiceTask.hasBeenExecuted()" which checks if the two tasks have been executed in the past is not correct, since it becomes true the first time the activities ShipProduct and SendInvoice were executed and cannot reflect any new execution after that. We need a different expression to represent that new executions have occurred that have not yet triggered an execution of ConfirmPO. This will be discussed in detail later in the next section.

Another factor that needs to be considered is the presence of invisible transitions, i.e., transitions without associated activity in the real world. For such invisible transitions no stage will be generated. Therefore, in order to compose the guard sentries, only visible pre-transitions should be considered. Thus we need to backtrack in the PN until we reach a visible transition and "collect" the relevant conditions of the branches we traverse. As an example, consider the transition DetermineSuppliers in Fig. 1. It can fire multiple times - at first when CreateWO has been executed and then every time the invisible pre-transition represented by a black rectangle fires. We backtrack to find the pre-places of the invisible transition and their pre-transitions. Here we determine that the only such pre-transition is ReceiveSupplResponse and this branch has an associated condition - we can only take this branch if the supplier rejects an order and a new supplier has to be determined.

With all these considerations in mind, the resulting guard sentry can become more complex and partly lose its advantage of being able to give intuition about how the execution of one task influences the execution of others. In order to solve

this problem, we apply methods for decomposing the expression into multiple simpler sentries which are then assigned to separate guards of the same stage. The composition and decomposition of guard sentries will be described more precisely in the next section.

Let t_o be the "origin", i.e., the (visible) transition for which we compose a guard. At a more abstract level the proposed method for generating guard sentries for the stage of t_o proceeds as follows:

Step 1: Find the relevant branch conditions and the pre-transitions whose execution will (help) trigger the execution of t_o.

Step 2: Decompose into groups that can be represented by separate guards.

Step 3: For each group, determine the appropriate format of the sentry and generate its expression.

5   Guard Sentries Generation

Our approach for achieving step 1 is inspired by the research presented in [10] for translating BPMN models and UML activity Diagrams into BPEL. It generates so-called precondition sets for all activities which encode possible ways of enabling an activity. Next, all the precondition sets with their associated activities, are transformed into a set of Event-Condition-Action (ECA) rules.

Before giving the precise definitions of the approach proposed here, we first illustrate the intuition behind it by a couple of examples. Consider the transition CompletePO in Fig. 1. In order for it to be enabled and subsequently fire, there need to be tokens in both of its pre-places. Therefore the precondition for enabling CompletePO is a conjunction of two expressions, each of which related to one pre-place and representing the fact that there is a token in this pre-place. This token could come from exactly one of the pre-transitions of this place. Consider for example the transition AssembleProduct. It has one pre-place which has two pre-transitions. Therefore the precondition here is a disjunction of two expressions each related to the firing of one pre-transition.

Thus the general form of the composed expression is a conjunction of disjunctions of expressions. These expressions, however, can themselves be conjunctions of disjunctions. This happens when a pre-transition is invisible (not observable in reality) and we need to consider recursively its pre-places and pre-transitions. The building blocks of the composed expression are expressions each of which corresponds to the firing of one visible transition t that can (help) trigger the firing of the transition in focus t_o (the "origin"). We denote each of these building blocks by $prcExpression(t, t_o)$ for a transition t with respect to t_o and they will be discussed in the next section.

Furthermore, the presence of a token in a pre-place is not a guarantee that a transition will fire. In the case of AssembleProduct, a token in its pre-place enables two transitions, AssembleProduct and SendCancellationInvoiceCust, but only one will fire. Which one is determined by conditions associated with each outgoing arc of the place. These conditions are domain-specific and, in the following, we assume that these conditions are given - they can be provided by

the user or mined from the logs using existing tools such as the decision miner from [11]. Therefore, the general form of the composed expression should have these conditions added to the conjunction.

The following more precise definitions reflect these intuitions on the general form of the expression and its recursive nature. By $prc(t_o)$ we denote the composed expression (of "pre-conditions") of the guard sentry for a stage/transition t_o. Let $IA(t_o)$ be the set of incoming arcs in t_o and let $cond_a$ be a condition associated with the arc a if the connected pre-place is a decision point (i.e., a place with multiple outgoing arcs) or *true* if it is not a decision point (no condition). Also, *init* denotes the specific expression of the event of the creation of the artifact instance, e.g. "onCreate()", $PT(a)$ is the set of pre-transitions t_p connected to the pre-place of the arc a.

We can then define $prc(t_o)$ using a recursive definition as follows:

$$prc(t_o) = \bigwedge_{a \in IA(t_o)} P_a \wedge cond_a$$

where P_a is defined as:

$$P_a = \begin{cases} init & \text{if } P_a \text{ is the initial place,} \\ \bigvee_{p \in PT(a)} T_p & \text{otherwise.} \end{cases}$$

T_p, in turn, is defined as follows:

$$T_p = \begin{cases} prcExpression(t_p, t_o) & \text{if } t_p \text{ is a visible transition,} \\ prc(t_p) & \text{if } t_p \text{ is an invisible transition.} \end{cases}$$

Here, as mentioned earlier, $prcExpression(t_p, t_o)$ is the specific expression that will be added to the sentry condition for each relevant visible transition t_p with respect to the "origin" t_o. Their format will be discussed in the next section.

The expression for $prc(t_o)$ can be represented in a tree structure in a straightforward way. The internal nodes of the tree represent logical operators ("and" or "or") with are applied on their child branches. The leaves represent either transitions that need to fire (which will be represented in the guard sentry by an expression $prcExpression(t_p, t_o)$ for the specific transition t_p in the leaf) or decision point conditions that need to be true in order the "origin" transition t_o to be able to fire. In the following we use the words tree and expression interchangeably since, in this context, they represent the same information.

An example of such a tree is given in Figure 2 constructed for the transition AssembleProduct. Looking at the model in Figure 1 we can see that AssembleProduct can only fire if there is a token in its pre-place and the condition associated with the connecting arc is true. We denote this condition here as Condition 1. The token can arrive from two possible transitions - ReceiveSupplCancellationResponse or the invisible transition represented as a black rectangle. We need to traverse back from the invisible transition and find out that it can only fire if the transition ReceiveSupplResponse fires and the condition associated with the connecting arc is true (we denote this condition by
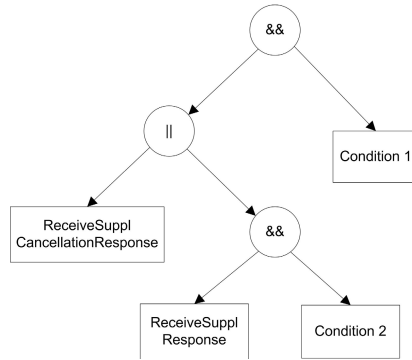
Fig. 2. An example of an expression tree which will be used to generate the guard(s) for stage AssembleProduct
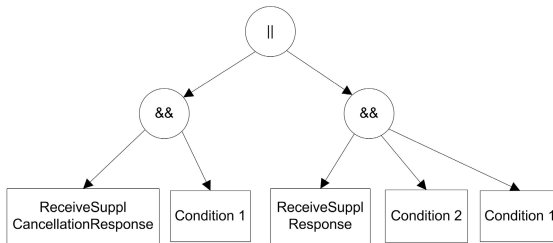

Fig. 3. The expression tree for stage AssembleProduct in DNF

Condition 2). This analysis results in the tree in Figure 2. The leaves of the tree are named by the corresponding transition or condition and, in fact, represent the specific expression for that transition/condition. However we delay the exact formulation of the expressions until the tree is built and analyzed, as will be described in the next section.

As mentioned earlier, an intermediate step of the algorithms decomposes $prc(t_o)$ into several expressions which then are used to generate separate guards of the stage. Since $prc(t_o)$ is a logical formula, we can convert it into Disjunctive Normal Form and assign each conjunction to a separate guard sentry.

After converting the example tree from Figure 2 into DNF, we now have the tree in Figure 3. Each child of the root node will generate one separate guard - here we have two guards. Intuitively the first guard tells us that the stage will open if task ReceiveSupplCancellationResponse was executed and Condition 1 is true. Similarly, the second guard tells us that the stage will open if task ReceiveSupplResponse was executed and both Condition 1 and Condition 2 are true.

As a final step, the $prcExpression(t_p, t_o)$ for the leaves of the tree are assigned as discussed in the next section.

6   Formats for Pre-condition Expressions

In this section we look into the expressions $prcExpression(t_p, t_o)$ in more details and define their format. Their assignment is delayed until the end, after $prc(t_o)$ is composed and, if needed, decomposed into separate sentries. Only then it can be decided which format each expression should take. We consider two possible formats for the expression of $prcExpression(t_p, t_o)$ depending on the context as discussed below.

The most simple case is when $prc(t_o)$ contains only one transition t_p with its expression $prcExpression(t_p, t_o)$ and $init$ is not present in $prc(t_o)$. Then $prcExpression(t_p, t_o)$ can be replaced by the event corresponding to the finished execution of the activity of t_p, we denote this by "on t_p executed". It can be expressed using the event of achieving the milestone of the stage of t_p or, alternatively, the closing of that stage among other options.

For example, for t_o = ReceiveSupplCancellationResponse the expression tree contains only one leaf corresponding to the transition t_p = SendRequestToSuppliers, i.e., the only way to enable t_o is by a token produced by t_p and this token cannot be consumed by another transition. Then the expression for t_p and t_o will be $prc(t_o) = prcExpression(t_p, t_o) =$ "on t_p executed".

If this is not the case, i.e., multiple transitions are present, then a more complex version of the expression needs to be included since we cannot use more than one event in the sentry. This form of the expression is discussed in the following paragraphs.

We introduce the following notation: for transitions t_1 and t_2, $t_1 \xrightarrow{\tau} t_2$ iff there exists a directed path in the graph of the net from t_1 to t_2 containing no visible transitions other than t_1 and t_2.

We now define the following set of "alternative" transitions to t_o, i.e., visible transitions that are connected to a place on the path from t_p to t_o:

$$Alt(t_p, t_o) = \{t \mid \exists \text{ place } p : t_p \xrightarrow{\tau} p \xrightarrow{\tau} t_o \ \wedge \ p \xrightarrow{\tau} t\}.$$

$Alt(t_p, t_o)$ are the set of transitions that "compete" with t_o for the token produced by t_p. Therefore in order to represent the situation when a token is present in the pre-place of t_o and the stage t_o should be opened we need to consider whether any of the "alternative" transitions have occurred (and "stolen" the token). Note that, according to this definition, t_o will also belong to the set.

Let us consider again the stage t_o = AssembleProduct and the expression tree in Fig. 3. Here we can use the simple format of the expressions for each leaf since in each branch there is only one transition. However for illustration purposes we assume that more that one transition was present in each branch and we need to use the more complex format for the expressions as follows. For the leaf t_p = ReceiveSupplCancellationResponse, $Alt(t_p, t_o)$ = {SendCancellationInvoiceCust, AssembleProduct}. Looking at Fig. 1, we can see that the transition SendCancellationInvoiceCust is indeed an "alternative" to AssembleProduct in the sense that is can "steal" the token produced by the transition ReceiveSupplCancellationResponse in the connecting place. Similarly,

for t_p = ReceiveSupplResponse, $Alt(t_p, t_o)$ = {ReceiveCancellationRequest, SendCancellationInvoiceCust, AssembleProduct}.

Then we define the expression as follows:

$$prcExpression(t_p, t_o) = \bigwedge_{t_s \in Alt(t_p, t_o)} executedAfter(t_p, t_s).$$

Here $executedAfter(t_p, t_s)$ expresses the situation when there is a new execution of t_p which occurs after the last execution of t_s, meaning that it is relevant for triggering the opening of the stage of t_o. How this will be expressed in the specific implementation can vary. Here we show how this can be done using the state of a milestone (achieved or not) and the time a milestone was last toggled. In that case:

$$executedAfter(t_p, t_s) = m_p.achieved \ \wedge \ m_p.lastToggled > m_s.lastToggled.$$

In other words, the milestone m_p of t_p is achieved and it was last toggled after the milestone m_s of t_s. Here we rely on the fact that the milestone of a stage will be invalidated as soon as the stage is reopened. This is ensured by including an invalidating sentry for each milestone.

For example, for t_o = AssembleProduct, t_p = ReceiveSupplCancellationResponse and t_s = SendCancellationInvoiceCust,

$$prcExpression(t_p, t_o) = executedAfter(t_p, t_s) \ \wedge \ executedAfter(t_p, t_o) =$$
$$= m_p.achieved \ \wedge \ m_p.lastToggled > m_s.lastToggled \ \wedge$$
$$\wedge \, m_p.lastToggled > m_o.lastToggled,$$

in other words, ReceiveSupplCancellationResponse was executed after the last execution of SendCancellationInvoiceCust and after the last execution of Assemble Product, i.e., the token in the connecting place has not been consumed yet.

7   Conclusions and Future Work

This paper proposed a method for translating PNs to GSM models which allows to use existing process mining algorithms for discovering PNs from event logs and generating GSM models from them. This contributes significantly to solving the problem of mining artifact-centric models from event logs by generating the life cycles of artifacts.

Additionally, the information model can be built by considering the logs as well and extracting the data attributes for each event type of the artifact. Existing tools such as [11] can be used to mine data-dependent conditions for the guards based on the discovered information model.

Future work will also develop methods that allow to discover the interactions between artifacts and thus multi-artifact GSM models can be generated.

The method in this paper generates a flat model where no hierarchy of stages is used. Future work will also consider methods for stage aggregation. One possible solution is to use existing algorithms for process abstraction (e.g. [1,3]) for

business process models and translate the discovered process hierarchy to GSM stage hierarchy. For example the Refined Process Structure Tree [7] can be a first step to discovering such a hierarchy.

References

1. Bose, R.P.J.C., Verbeek, H.M.W., van der Aalst, W.M.P.: Discovering Hierarchical Process Models using ProM. In: Proc. of CAiSE Forum, CEUR Workshop Proc., vol. 734, pp. 33–40 (2011)
2. Cohn, D., Hull, R.: Business artifacts: A data-centric approach to modeling business operations and processes. IEEE Data Eng. Bull. 32, 3–9 (2009)
3. Günther, C., van der Aalst, W.: Mining activity clusters from low-level event logs, BETA Working Paper Series, WP 165, TU/e (2006)
4. Hein, J.L.: Discrete Structures, Logic, and Computability. Jones and Bartlett Publishers (2010)
5. Hull, R., Damaggio, E., Fournier, F., Gupta, M., Heath III, F(T.), Hobson, S., Linehan, M., Maradugu, S., Nigam, A., Sukaviriya, P., Vaculin, R.: Introducing the Guard-Stage-Milestone Approach for Specifying Business Entity Lifecycles. In: Bravetti, M. (ed.) WS-FM 2010. LNCS, vol. 6551, pp. 1–24. Springer, Heidelberg (2011)
6. Hull, R., et al.: Business Artifacts with Guard-Stage-Milestone Lifecycles: Managing Artifact Interactions with Conditions and Events. In: DEBS 2011, pp. 51–62 (2011)
7. Johnson, R., Pearson, D., Pingali, K.: The Program Structure Tree: Computing Control Regions in Linear Time. In: Proc. of the ACM SIGPLAN 1994 Conference on Programming Language Design and Implementation, pp. 171–185. ACM (1994)
8. Murata, T.: Petri nets: Properties, Analysis and Applications. In: Proc. of the IEEE, pp. 541–580 (1989)
9. Nigam, A., Caswell, N.S.: Business artifacts: An approach to operational specification. IBM Systems Journal 42(3), 428–445 (2003)
10. Ouyang, C., Dumas, M., Breutel, S., ter Hofstede, A.H.M.: Translating Standard Process Models to BPEL. In: Martinez, F.H., Pohl, K. (eds.) CAiSE 2006. LNCS, vol. 4001, pp. 417–432. Springer, Heidelberg (2006)
11. Rozinat, A., van der Aalst, W.M.P.: Decision Mining in ProM. In: Dustdar, S., Fiadeiro, J.L., Sheth, A.P. (eds.) BPM 2006. LNCS, vol. 4102, pp. 420–425. Springer, Heidelberg (2006)
12. van der Aalst, W., Barthelmess, P., Ellis, C., Wainer, J.: Proclets: A Framework for Lightweight Interacting Workflow Processes. Int. J. Cooperative Inf. Syst. 10(4), 443–481 (2001)
13. Verbeek, H., Buijs, J.C., van Dongen, B.F., van der Aalst, W.M.P.: Prom: The process mining toolkit. In: Proc. of BPM Demonstration Track. CEUR Workshop Proc., vol. 615 (2010)
14. Wilson, J.: Algorithms for obtaining normal forms of logical expressions. International Journal of Computer Mathematics 27(2), 85–90 (1989)