Deterministic Petri Net languages as Business Process Specification Language

Manu De Backer, Monique Snoeck
K.U.Leuven, Dept. of Applied Economic Sciences,
Naamsestraat 69, B-3000 Leuven, Belgium
{Manu.Debacker; Monique.Snoeck}@econ.kuleuven.be

Abstract

Today, a wide variety of techniques have been proposed to model the process aspects of business processes. The problem, however, is that many of these are focused on providing a clear graphical representation of the models and give almost no support for complex verification procedures. Alternatively, the use of Petri Nets as a business process modeling language has been repeatedly proposed. In complex business processes the use of Petri Nets has been criticized and the technique is believed to be unable to capture such processes in all aspects. Therefore, in this paper, we introduce the application of Petri Net language theory for business process specification. Petri Net languages are an extension to the Petri Net theory, and they provide a set of techniques to describe complex business processes more efficiently. More specifically, we advocate the application of deterministic Petri Net languages to model the control flow aspects of business processes. The balance between modeling power and analysis possibilities makes deterministic Petri Nets a highly efficient technique, used in a wide range of domains. The proof of their usability, as business process specification language, is given by providing suitable solutions to model the basic and more complex business process patterns [4]. Additionally, some points of particular interest are concisely discussed.

Keywords: Petri Net theory; Business Process Modeling; Verification

1 Introduction

Business Process Management does not only refer to the myriad of tasks and tools that are necessary to model, design, analyse and maintain business processes, but also to the tools that are indispensable to implement business processes in such way that they can be managed from a business perspective. Modeling business processes, essentially, comprises the creation of multiple models to capture all the aspects of business processes entirely. Most frequently described aspects are amongst others the control flow perspective, the data aspects and resource allocation. Traditionally, the focus of researchers and industry is mainly on the control flow aspects of business processes. For this, many different techniques and standards have been proposed, ranging from more formal techniques such as finite state machines and Petri Nets, to less formal ones such as BPMN and BPEL. However, many of these less formal techniques suffer from severe problems if they are evaluated on topics concerning verification. The formal techniques provide better verification methods, but often, these models are too complex to be comprehensible by the human experts that have to validate them.

Deterministic Petri Net languages could offer a technique with perfect verification capabilities and comprehensible models. Therefore, the purpose of this report is to examine how deterministic Petri Net languages can graphically and formally represent the main process patterns. For this we use the process patterns defined in [4]. For each pattern, we will show how deterministic Petri Net languages can implement the pattern at hand. Additionally, we will discuss for each pattern the limitations of our approach and give directions to implement the pattern properly.

The rest of the paper is structured as follows. First, we discuss some basic Petri Net theory. In the next section, we use this basic Petri Net theory to discuss the Petri Net language theory. In section 4, we respectively discuss the basic patterns, advanced branching constructs, structural patterns, and patterns involving multiple instances. Finally, in section 5 we conclude this paper and present some topics for further research.

2 Petri Net Theory

An ordinary Petri Net structure is a triple, $N = \{P, T, A\}$, where:

$$\begin{aligned}
&P = \{p_1, p_2, ..., p_n\} \text{ a finite set of places,}\\
&T = \{t_1, t_2, ..., t_m\} \text{ a finite set of transitions,}\\
&A \subseteq (P \times T) \cup (T \times P) \text{ is the flow relation,}\\
&(P \cap T = \emptyset) : \text{P and T are disjoint sets.}
\end{aligned} \tag{1}$$

Graphically, places are represented by circles and transitions are represented by boxes. The flow relation (A) is shown by the directed arcs between places and transitions, as illustrated in Figure 1.

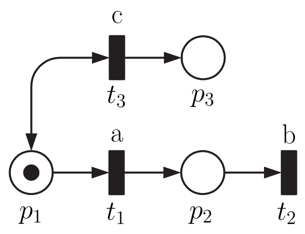

Figure 1: Example of a Petri Net

The preset and postset of a transition $t \in T$, called respectively input places and output places, are defined as ${}^\bullet t = \{p|(p,t) \in A\}$ and $t^\bullet = \{p|(t,p) \in A\}$. A marking of a Petri Net $N = (P, T, A)$ is a function from the set of places to the nonnegative integers $\mathbb{N}, \mu : P \to \mathbb{N}$. A partial marking is a function from the set of places to $\mathbb{N}^+ = \mathbb{N} \cup \omega$ thus $\mu : P \to \mathbb{N}^+$ with $\omega + c = \omega - c,\ c \leq \omega$ and $\omega \leq \omega$. A transition $t \in T$ is enabled in a marking μ if

$$\forall p_i \in {}^\bullet t : \mu(p_i) \geq |(p_i, t)|, i = 1...n. \tag{2}$$

where $|(p_i, t)|$ denotes the number of occurrences of (p_i, t) in A.

The firing of a transition t_j in a marking μ leads to a marking μ';

$$\mu'(p_i) = \mu(p_i) - |(p_i, t_j)| + |(t_j, p_i)|, i = 1...n. \tag{3}$$

This rule (3) is generally known as the firing rule. We write $\mu[t_j\rangle$ to denote that t_j may fire in μ, and $\mu[t_j\rangle\mu'$ to indicate that the firing of t_j in μ leads to μ'. In the same way, we write $\overline{\mu[t_j\rangle}$ to denote that t_j cannot fire in μ. Furthermore, the firing of a sequence of transitions (ρ) is defined as $\rho = t_1, t_2, ..., t_k$ such that $\mu_0[t_1\rangle\mu_1[t_2\rangle\mu_2 \cdots [t_k\rangle\mu_k$ which is abbreviated as $\mu_0[\rho\rangle\mu_k$. A marking μ is reachable if there exists a firing sequence ρ such that $\mu_0[\rho\rangle\mu$. The reachability set $R(N,\mu)$ of a Petri Net N with marking μ is the set of reachable markings from μ.

3 Petri Net Languages

In 1976, Hack [1] published a report on Petri Net languages where he stated that in many applications of Petri Nets it is the set of firing sequences generated by the net that is of prime importance. At this time it was proposed to treat Petri Nets like an automaton whose states are the markings of the Petri Net, and whose state-transition function expresses how and when transitions of the Petri Net can fire. This report was the start of an extensive research effort in Petri Net languages, which resulted in the definition of a wide range of Petri Net language families each having their own properties. This section introduces the basic concepts of Petri Net languages, for a more elaborate discussion the reader is referred to [3, 1, 2].

Basically, a Petri Net language is generated by a labeled Petri Net $PN = (N, \tau, \mu_0, F)$ with [3, 1, 2]:

$$\begin{aligned}
&N = (P, T, A) \text{ is a Petri Net,} \\
&\tau : T \to \Sigma \text{ a labeling of } T \text{ in the alphabet } \Sigma, \\
&\mu_0 \text{ is the initial marking,} \\
&F \text{ is a set of final markings.}
\end{aligned} \tag{4}$$

The labeling function τ assigns to each transition a label from the alphabet Σ. A finite set of symbols is called a word or a string (w). A language (L) is a set of strings from Σ. Σ^* is the Kleene star operation on the alphabet Σ, which is the concatenation of none, one, two or any countable number of symbols of the alphabet Σ.

The initial marking (μ_0), the labeling function (τ) and the definition of the set of final markings (F) play a crucial role in the generation of Petri Net languages. When any of these are changed the generated language will change accordingly. Consequently, a single ordinary Petri Net can generate a whole range of languages just by changing the begin marking, labeling function or the final marking set.

The definition of the initial marking can take different forms: a single marking, a single marking with only one token in a start place, a set of initial markings, etc. Note that these three definitions are in fact equivalent.

Generally, four alternative labeling functions are considered in the literature [1, 3, 5]. First of all, a free-labeled Petri Net is a Petri Net where all transitions are labeled distinctly, i.e if $\tau(t_i) = \tau(t_j)$, then $t_i = t_j$. Secondly, the class of λ-free Petri Net languages allow a non-distinct labeling of the transitions but no empty transitions are allowed, i.e. $\forall t_i \in T : \tau(t_i) \neq \lambda$. Furthermore, an even more relaxed constraint on the labeling function allows empty (λ) labeled transitions, meaning that the labeling function τ is partial ($\exists t_i \in T : \tau(t_i) = \lambda$). In [5], Vidal-Naquet showed that there is a fourth labeling function which was overseen by Hack and Peterson. The deterministic labeling function has the additional property that at each marking and for each label, at most one transition with this label is firable, i.e. $\forall \mu_i \in R(N, \mu_0)$ and $\forall t, t' \in T$: $(\tau(t) = \tau(t')$ and $\mu_i[t\rangle$ and $\mu_i[t'\rangle) \Rightarrow t = t'$. According to these different definitions of the labeling function four different types of languages can be defined, respectively referred to as L^f, L, L^λ and L^{det}.

A third manner to alter the generation of a Petri Net language is by changing the definition of the set of final markings (F). Generally, four variations of the set of final markings are considered.

Given a labeled Petri Net $PN = (N, \tau, \mu_0, F)$, the *L-type* Petri Net language is:

$$L(PN) = \{\tau(\rho) \in \Sigma^* | \rho \in T^*, \mu_0[\rho\rangle\mu, \mu \in F\} \tag{5}$$

the *T-type* Petri Net language is:

$$T(PN) = \{\tau(\rho) \in \Sigma^* | \rho \in T^*, \mu_0[\rho\rangle\mu, \forall t_i \in T : \overline{\mu[t_i\rangle}\} \tag{6}$$

the *P-type* Petri Net language is:

$$P(PN) = \{\tau(\rho) \in \Sigma^* | \rho \in T^*, \mu_0[\rho\rangle\} \tag{7}$$

the *G-type* Petri Net language which is also referred to as the weak language is:

$$G(PN) = \{\tau(\rho) \in \Sigma^* | \rho \in T^*, \mu_0[\rho\rangle\mu, \mu \geq \mu' \ for\ some\ \mu' \in F\} \tag{8}$$

If we consider the Petri Net of Figure 1 and a final marking set $F = \{(0, 1, 0)\}$ we can summarize the different generated languages as in Table 1.

Language Type	Language
L-type	$L(PN) = \{a\}$
T-type	$T(PN) = \{c^n ab \| n \geq 0\}$
P-type	$P(PN) = \{c^n, c^n a, c^n ab \| n \geq 0\}$
G-type	$G(PN) = \{c^n a \| n \geq 0\}$

Table 1: Different languages generated by the Petri Net in Figure 1

4 Petri Net Languages as Business Process Specification Language

In the previous sections, we have discussed Petri Net and Petri Net language theory. In this section, the use of Petri Net languages for business process modeling is further considered. Essentially, the control flow aspects of a business process define a set of sequence constraints on a set of tasks that need to be executed in the process. Therefore, we will define the alphabet of a labeled Petri Net as the set of activities of the process. Next, the labeling function defines how the symbols in the alphabet are projected on the transitions. In this way, the Petri Net structure defines a set of sequence constraints on the tasks of the business process, i.e. the Petri Net generates a language over the tasks of the process. Additionally, we require the labeling function to be deterministic.

Further, a Petri Net language requires the specification of a set of final markings and a indication of how these final marking set is used e.g. L-type, G-type, T-type or P-type. In our case, a business process is described as an L-type deterministic Petri Net language.

4.1 Basic Control Patterns

4.1.1 Pattern 1: Sequence

A sequence pattern contains two or more ordered activities that are performed sequentially, i.e. an activity starts after a previous activity has completed. This pattern is easily implemented by means of the basic Petri Net constructs: for each activity a transition is created and the transitions are connected with each other by means of arrows and places. The sequence flow direction is determined by the flow relation, e.g. the arrows in Figure 2.

Additionally, to define a Petri Net language, we need to specify the labeling function, the begin and the final marking. Obviously, the labeling function shall rename the transitions with the names of the activities they represent. We define the begin and the set of final markings for this Petri Net as follows: $\mu_0 = (1, 0, 0)$ and F={(0,0,1)}. The labeled Petri Net PN=(N,τ,μ_0,F) defines the language L(PN)={AB}, i.e. activity B is only executed after the completion of activity A.

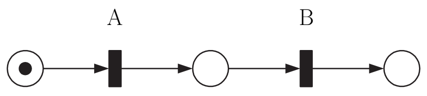

Figure 2: The sequence pattern

Example: An order process, for example, usually contains the following behavior or some variant thereof: first an order is created ($create\_order$), then the order is processed ($process\_order$), fabricated ($fabricate\_order$) and shipped ($ship\_order$).

4.1.2 Pattern 2: Parallel Split

The parallel split pattern is defined as being a mechanism that allows activities to be executed concurrently. The single thread of control is split into two or more threads, which means that the activities can be executed at the same time or in any order. In fact, the parallel split is used when there is no sequence constraint defines on a set of activities. This pattern is also easily implemented by means of the basic Petri Net constructs: a transition is connected to multiple (output)places, i.e. the firing of this transition will enable multiple transitions at the same time, e.g. the firing of transition C enables transitions A and B, see Figure 3.

Again, a Petri Net language is defined by specifying the labeling function and the begin and final marking. There are no special requirements in the definition of the begin marking μ_0=(1,0,0,. . . ,0) and the set of final markings F={(0,0,0,. . . ,1)}. For the labeling function, however, we have to be careful and ensure that there are no duplicate labels in each thread of control, as this would break the determinism requirement (cf. supra). At first sight, this additional requirement seems very stringent, but in fact in business process terms it is a plausible restriction. The possible simultaneous execution of the same activity has no meaning. Moreover, whenever such a construction seems convenient, it in fact turns out that, in business terms, the activities have a different connotation. The language defined by the Petri Net in Figure 3 is L(PN)={CAB. . . ,CBA. . . }.

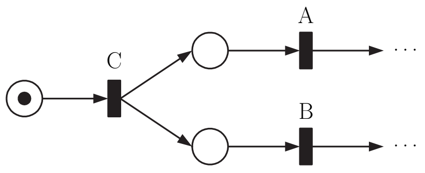

Figure 3: The parallel split pattern

Example: The alarm procedure in a highly toxic plant can be described as follows: if a problem occurs then activate the alarm (*activate_alarm*). Next, the police (*notify_police*) as well as the fire-department (*notify_fire*) should be notified.

4.1.3 Pattern 3: Synchronization

The synchronization pattern is used to merge the different threads that are started by a parallel split. This means that all the threads of the parallel split must be completed before the process can continue. In Petri Net terminology the synchronization pattern is implemented by connecting the places of each concurrent thread with one new transition. This means that each parallel thread needs to finish (add a token in the place) before the process can continue with the next activity, e.g. transitions A and B need to fire to enable transition C, see Figure 4.

The Petri Net language that we need to define for this pattern has no special features, and can be specified as follows: μ_0=(1,...,0,0,0) and the set of final markings F={(0,...,0,0,1)}. Thus, L(PN)={...ABC,...BAC}

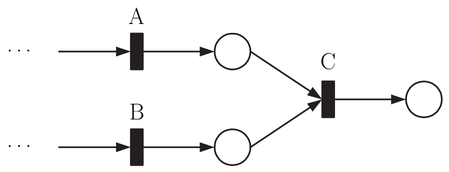

Figure 4: The synchronization pattern

Example: The alarm procedure could continue as follows: after the notification of police ($notify\_police$) and fire department ($notify\_fire$) shut down the electricity ($shutdown\_electricity$) and leave the building ($leave\_building$).

4.1.4 Pattern 4: Exclusive Choice

This pattern defines a place in the process where exactly one of multiple exclusive threads is executed. The exclusive choice pattern supports conditional behavior and is also directly supported by basic Petri Net constructs. Connecting several transitions with one place results in a situation where multiple transitions are enabled but the firing of one will disable the others, e.g. if transition A is fired, transitions B and C are disabled, see Figure 5.

The begin marking and set of final markings are defined as follows: μ_0=(1,0,0,0,...,0) and F={(0,0,0,0,...,1)}. Some special attention is needed for the labeling function as the exclusive choice pattern defines a set of transitions that are enabled at the same time. An additional restriction is specified: each activity in the exclusive choice pattern should be unique, i.e. the labeling of each transition in the pattern should be unique. The case where transitions with the same label are allowed breaks the determinism constraint. This, again, is a plausible constraint in business process terms. The language defined by the Petri Net is L(PN)={A...,B...,C...}.

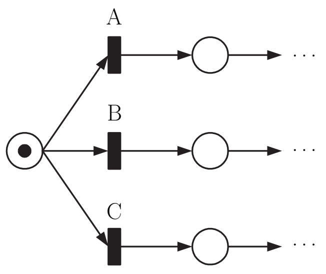

Figure 5: The exclusive choice pattern

Example: If there is a fire then activate the fire-extinguishers ($activate\_fireExt$). If there is a leak in a highly toxic tank, then close the windows ($close\_windows$).

4.1.5 Pattern 5: Simple Merge

The simple merge pattern is used to bring together the paths of an exclusive choice pattern. This means that after the execution of one of the paths of the exclusive choice exactly one and the same activity needs to execute. This pattern is accomplished by connecting the last transitions of the different paths with the same place, the execution of the A, B or C branch will always enable transition D, see Figure 6.

No additional constraints are required to define this pattern as a deterministic Petri Net language, we just have to define the begin marking and the set of final markings as follows: μ_0=(1,...,0,0,0,0,0) and F={(0,...,0,0,0,0,1)}. There is no restriction on the labeling function. L(PN)={...AD,...BD, ...CD}.

Example: After the activation of the fire-extinguishers ($activate\_fireExt$) or the closing of the windows ($close\_windows$) we notify the fire department ($notify\_fire$).

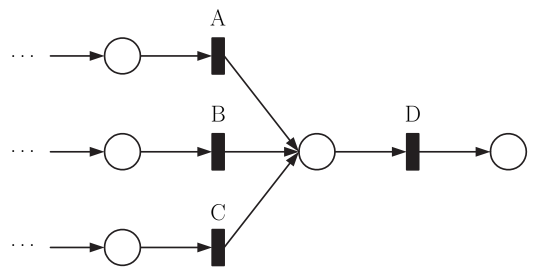

Figure 6: The simple merge pattern

4.2 Advanced Branching and Synchronization Patterns

4.2.1 Pattern 6: Multiple Choice

Compared with the exclusive choice pattern, the multiple choice pattern allows multiple paths to be executed. The complex nature of this pattern makes it impossible to define it by means of a simple Petri Net construct. The most obvious way to implement this pattern is by combining a parallel split pattern and the synchronization pattern. However, the problem with this solution is that the synchronization pattern requires all concurrent threads to finish before the process can continue. Therefore, we propose a solution based on the notion of *toggle* transitions. A toggle transition is represented by means of two transitions with the following labels and meaning. A first transition represents the execution of the activity (e.g. A), while the other represents the non-execution of the same activity (e.g. NOT A). These toggle transitions are implemented through an exclusive choice pattern in combination with the simple merge pattern. To complete the construction of the multiple choice pattern, the set of toggle transitions are then combined by means of the parallel split pattern en synchronized through the synchronization pattern.

Except for the additional requirement that is needed to implement the parallel split pattern (cf. supra), there are no special requirements needed. We define the begin and final marking as follows: μ_0=(1,0,0,0,0,0) and F={(0,0,0,0,0,1)}. The generated language is L(PN)={ABCD,ACBD,ABD,ACD,AD}.

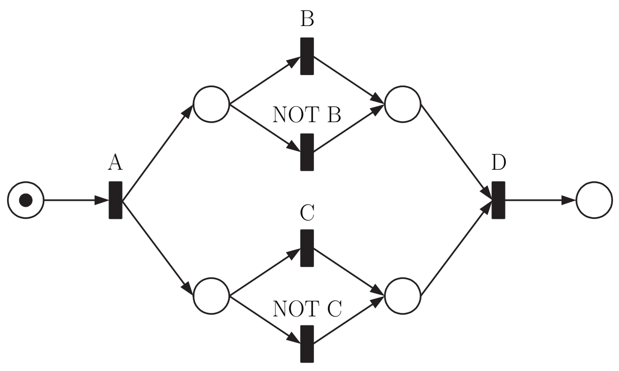

Figure 7: The multiple choice pattern

Example: The alarm procedure in a highly toxic plant can be described as follows: if a problem occurs then activate the alarm ($activate\_alarm$). Next, depending on the severity of the problem the police ($notify\_police$) and/or the fire-department ($notify\_fire$) should be notified.

4.2.2 Pattern 7: Synchronizing Merge

The synchronizing merge pattern takes care of the synchronization after the execution of a multiple choice pattern. If more than one thread from the multiple choice pattern is executed then this pattern will take care of the synchronization. If only one path of the multiple choice pattern is executed then the alternative branches will converge without synchronization. This pattern states that independent from the number of threads that are executed from a multiple choice pattern the following activity is executed just once. This pattern is easily implemented because of the way we have implemented the multiple choice pattern, see Figure 8.

There are no additional requirements for implementing the pattern as a deterministic Petri Net language. The begin marking and the set of final markings are specified as follows: μ_0=(1,0,0,0,0,0) and F={(0,0,0,0,0,1)}. This Petri Net will generate the language L(PN)={ABCD,ACBD,ABD,ACD,AD}.

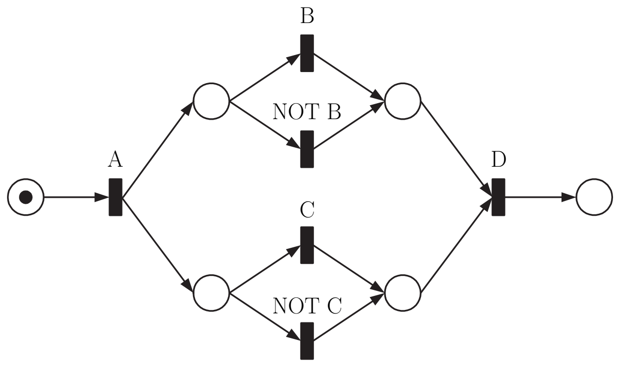

Figure 8: The synchronizing merge pattern

Example: The alarm procedure could continue as follows: after the notification of police ($notify\_police$) and/or fire department ($notify\_fire$) shut down the electricity ($shutdown\_electricity$) and leave the building ($leave\_building$).

4.2.3 Pattern 8: Multiple Merge

The multiple merge pattern, can also be used in combination with the multiple choice pattern, but this pattern does not provide synchronization of the executed threads i.e. if n threads are executed, possibly concurrently, the activity following the merge is executed n times. Figure 9 depicts a possible implementation of the multiple merge pattern, if activity B and C are executed then activity D will be executed twice.

No additional requirements are needed to implement this pattern as a deterministic Petri Net language. We define the begin marking and the set of final markings as follows: μ_0=(1,0,0,0,0) and F={(0,0,0,0,1)}. The language generated by this Petri Net is L(PN)={ABCDD,ABDCD,ACDBD,ACBDD,ABD,ACD}.

Example: A paper reviewing system sends the papers to three reviewers. Depending on the number of reactions the activity $request\_review$ is executed once or multiple times. For instance, if two reviewers react then the system invokes the activity $request\_review$ two times.

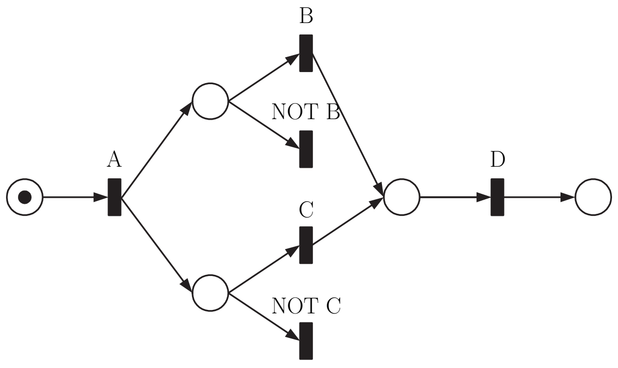

Figure 9: The multiple merge pattern

4.2.4 Pattern 9: Discriminator

The discriminator pattern is used to merge a set of concurrent threads, instead of waiting for all activities to finish, the discriminator pattern allows the process to continue when the first thread has finished. The finishing of the other threads is ignored. Such a complex pattern is not easily implemented using Petri Nets. The pattern is used in combination with the parallel split pattern which enables a set of concurrent execution threads.

Petri Net languages offer an efficient solution for implementing the discriminator pattern. The difficult part of the implementation is about preventing the subsequent activity from firing multiple times. This kind of restriction can be imposed by defining a proper set of final markings. More specifically, we explicitly specify that the final marking must have two tokens in the place following the parallel split. This will restrain the Petri Net from firing the activity multiple times. The begin marking remains the same, i.e. μ_0=(1,0,0,0,0,0). The final marking however will be specified as follows: F={(0,0,0,0,2,1)}. This means that the Petri Net language defined by this Petri Net is: L(PN)={ABCDE,ABDCE,ABECD,ABCED,ABDEC,ABEDC,ACBDE,ACDBE,ACEBD, ACBED,ACDEB,ACEDB,ADCBE,ADBCE,ADECB,ADCEB,ADBEC,ADEBC}

Example: If an order arrives at an online shop, the system will check the stock to see if the order can be delivered. If the stock is insufficient, it will automatically contact multiple wholesalers to replenish the stock. The wholesaler that responds first with an

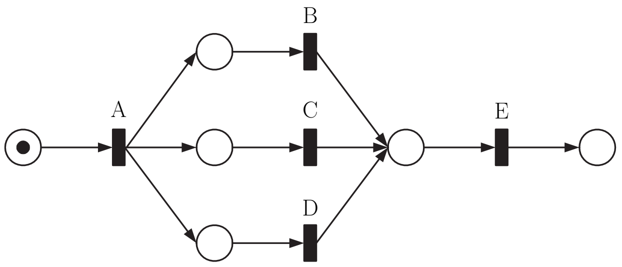

Figure 10: The Discriminator pattern

acceptable price offer will get the deal. The answers of the others are ignored.

4.2.5 Pattern 9a: N-out-of-M-join

The N-out-of-M-join pattern describes a situation where the subsequent activity of a parallel split pattern is executed once when n activities have finished. The execution of the rest of the threads $(m - n)$ is ignored. This pattern is implemented in the same way as the discriminator pattern. By varying the set of final markings of a given Petri Net the N-out-of-M-join pattern is easily implemented. The use of this technique enables us to easily implement different variations of the pattern, e.g. the next activity is executed once if 3-out-of-5 or 2-out-of-5 activities are executed. The following example shows a situation where two of the three activities should be executed before the next activity is enabled, see Figure 11.

The begin marking is specified as before: μ_0=(1,0,0,0,0,0). The set of final markings is altered as follows: F={(0,0,0,0,1,1)}. The Petri Net language defined by this labeled Petri Net is: L(PN)={ABCDE,ABCED,ABDCE,ABDEC,ACBDE,ACBED,ACDBE, ACDEB,ADBCE,ADCBE,ADBEC,ADCEB}.

Example: To replenish the stock, a company contacts three wholesalers. On receiving two offers the company can compare the offers and take a decision. The third offer is then ignored.

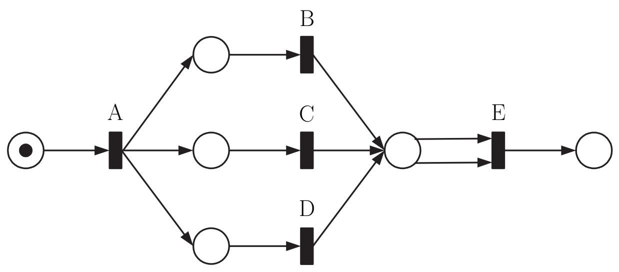

Figure 11: The N-out-of-M-join pattern

4.3 Structural Patterns

4.3.1 Pattern 10: Arbitrary Cycles

The arbitrary cycles pattern supports the structure where a set of activities can be executed repeatedly. This pattern is easily implemented by means of the basic Petri Net constructs. The pattern can be implemented with the exclusive choice pattern where one or more of the branches loops back to re-execute a set of activities.

No special additions are required to define the pattern by means of a Petri Net language. The begin marking is defined as μ_0=(1,0,0,0) and the set of final markings can be specified as F={(0,0,0,1)}. The language generated by this Petri Net is L(PN)=(ABC,ABDBC,ABDBDBC,ABDBDBDBC,...).

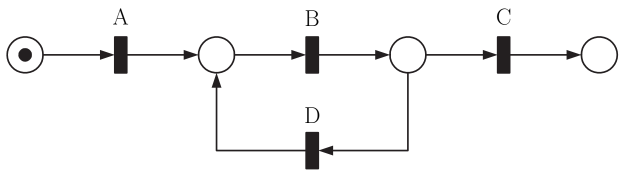

Figure 12: Arbitrary cycles

Example: Imagine a situation where an evaluation activity triggers one of the following activities: $eval\_failed$ and *eval_passed.* The failed activity will reactivate the task $produce\_order$.

4.3.2 Pattern 11: Implicit Termination

The implicit termination pattern means that a given process should be terminated if at a given state in the process, no activity is enabled and no activity can get enabled and at the same the process is not in deadlock. We believe that this pattern is not so important as the other patterns as it does not describe a sequence constraint on a set of activities. Therefore, we will not discuss this pattern in the paper. Note that the absence of tokens in the Petri Net, i.e. no transitions are enabled, is the Petri Net counterpart of the implicit termination pattern.

4.4 Patterns Involving Multiple Instances

The advantages of the Petri Net language approach will become clear in more complex scenarios that we will discuss here. One of the disadvantages of using pure Petri Net constructs for business process modeling, is their inability to model control flows where multiple case instances are involved.

4.4.1 Pattern 12: Multiple Instances Without Synchronization

This pattern is best explained by means of an example:

Example: The process control flow describes the following behavior: for each customer the ordering process is started by creating an order (*cr_order*) then for each product the customer orders, an orderline (*cr_orderline*) is added to the order. These orderlines can be changed (*ch_orderline*) during the process and other orderlines can be added to the order. Once all the orderlines (*end_orderline*) are added to the order, the system will be able to close the order and calculate the total amount of the order. This process behavior is also called interleaving and is considered to be a very difficult to model construction.

This process behavior is impossible to model by means of the basic Petri Net constructs. However, the use of Petri Net languages can yield important opportunities. The following Petri Net describes a particular implementation of the interleaving construct, see Figure 13. The definition of the begin and final marking of the Petri Net is as follows: μ_0=(1,0,0,0,0,0) and F={(0,0,0,ω,0,1)}. This final marking adds supplementary sequence constraints on the activities, for instance the firing sequence *cr_order.cr_orderLine.cancel_order.archive* is supported by the Petri Net but it is not a word of the Petri Net language as the marking reached after this firing sequence is (0,0,1,0,01) which is not an element of F.

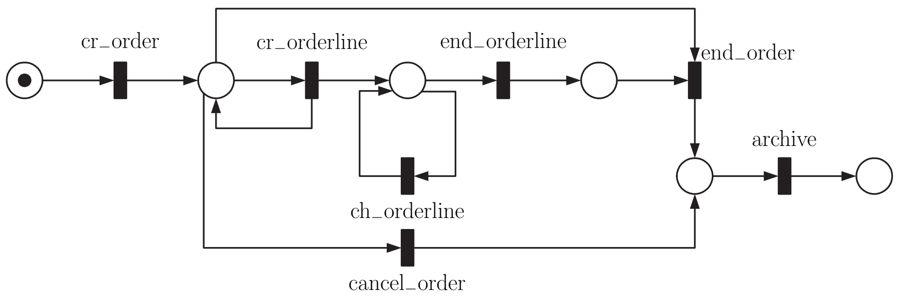

Figure 13: A process flow with multiple instances

4.4.2 Pattern 13: Multiple Instances With a Priori Design Time Knowledge

This pattern describes the behavior where for each process instance an activity is executed multiple times. The number of executions is determined in advance. This pattern can be implemented in two ways. The first option is to implement this pattern by means of a parallel split pattern. Once all activities are completed they can be synchronized using the standard synchronizing construct, see Figure 14. In this case, we have to remind the additional constraint of applying the parallel split pattern, i.e. the concurrent threads should define unique activities.

Example: The shipping of hazardous material requires three different authorizations. These authorizations should be provided by several different instances: the government, the environmental council and the transportation firm. Therefore, we can rename these activities as: *gov_auth*, *env_auth* and *trans_auth*, which can be implemented efficiently by means of the parallel split.

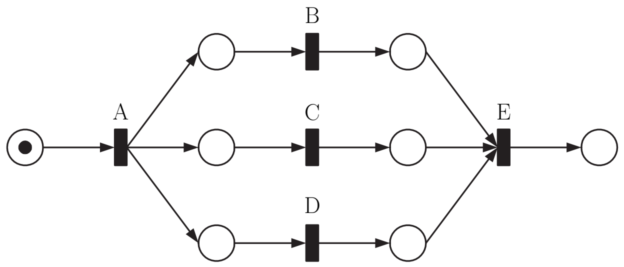

Figure 14: A process flow with multiple instances, version 1

In the case that a specific activity needs to be executed multiple times we can also implement this pattern in the following way, see Figure 15.

Example: An insurance claim handling system will demand three experts to formulate a conclusion about the fraud rate of a certain claim. Since it is impossible to select the three experts in advance, it is of no point to split up the activity.

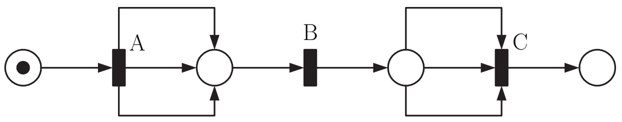

Figure 15: A process flow with multiple instances, version 2

4.4.3 Pattern 14: Multiple Instances With a Priori Runtime Knowledge

This pattern is used when depending on certain case attributes an activity is executed multiple times. Once all the instances of the activity are finished the following activity is executed. The specification of this pattern requires the definition of a set of final markings. F should be defined as follows: F=$\{(0,0,\omega,1)\}$. Moreover, the set of final markings can be altered dynamically, i.e. if the number of times an activity needs to be executed is known, we can redefine the final markings based on this new information. The Petri Net language of Figure 16 defines a process where activity A can be executed multiple times, but activity C is only allowed to execute if activity B is executed for every instance of A. Finally, the execution of C disables activity A.

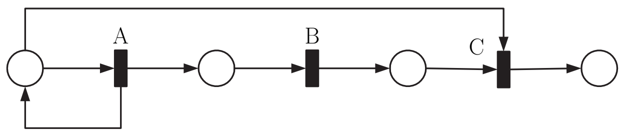

Figure 16: A process flow with multiple instances

Example: A flight booking system executes the activity $book\_flight$ multiple times if the trip involves multiple flights.

4.4.4 Pattern 15: Multiple Instances Without a Priori Runtime Knowledge

An example describes this pattern more precisely.

Example: A batch order requires multiple deliveries. At no time in the process it is, however, clear how many deliveries are necessary. Therefore, the activity *deliver* is executed multiple times, until all products are delivered. For example, an order of 200 cars, can only be executed in multiple deliveries.

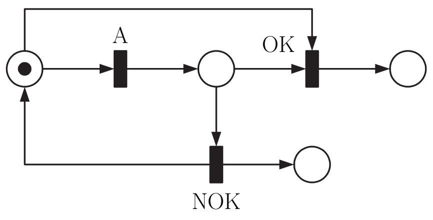

Figure 17: A process flow with multiple instances

5 Conclusions

In this paper, we have demonstrated that deterministic Petri Net languages are an efficient alternative for most business process specification languages. By means of a set of generally approved patterns we showed that, in all cases, the approach generated the patterns in a straightforward manner and, in most cases, Petri Net languages yielded highly comprehensible models. However, sometimes, the approach required the specification of an additional constraint on the labeling function, this, by no means influenced the modeling power of the approach. In contrary, the Petri Net languages approach allows to construct very complex behavior in a very efficient way.

Additionally, we believe that Petri Net language theory, compared to many other business process modeling approaches, supports a myriad of efficient verification techniques. Clearly, our approach benefits substantially from the many research efforts in Petri Net theory. Of course, this approach also requires some new specific analysis techniques but the development of new techniques is highly supported by the formal basis of Petri Net theory. Therefore, an interesting topic for further research would be to conceptualize and implement, on the one hand, a set of analysis techniques and, on the other hand, investigate the use of these techniques on real-life business process models.

References

[1] M. Hack. Petri net languages. Technical report, Massachusetts Institute of Technology, 1976.

[2] M. Jantzen. Language theory of petri nets. In Brauer, W., Reisig, W., and Rozenberg, G., editors, *LNCS: Petri Nets: Central Models and Their Properties, Advances in Petri Nets 1986, Part I, Proceedings of an Advanced Course, Bad Honnef, September 1986*, volume 254, pages 397–412. Springer-Verlag, 1987.

[3] J. L. Peterson. *Petri Net Theory and the Modeling of Systems.* Prentice-Hall, 1981.

[4] W. M. P. van der Aalst, A. H. M. ter Hofstede, B. Kiepuszewski, and A. P. Barros. Workflow patterns. *QUT Technical report, FIT-TR-2002-02*, 2002.

[5] G. Vidal-Naquet. Deterministic languages of petri nets. In Girault, C. and Reisig, W., editors, *Informatik-Fachberichte 52: Application and Theory of Petri Nets. Strasbourg, Sep. 23-26, 1980, Bad Honnef, Sep. 28-30, 1981*, pages 198–202. Springer-Verlag, 1982.