Chapter 3
Process Modeling and Analysis

The plethora of process modeling notations available today illustrates the relevance of process modeling. Some organizations may use only informal process models to structure discussions and to document procedures. However, organizations that operate at a higher BPM maturity level use models that can be analyzed and used to enact operational processes. Today, most process models are made by hand and are not based on a rigorous analysis of existing process data. This chapter serves two purposes. On the one hand, preliminaries are presented that will be used in later chapters. For example, various process modeling notations are introduced and some analysis techniques are reviewed. On the other hand, the chapter reveals the limitations of classical approaches, thus motivating the need for process mining.

3.1 The Art of Modeling

In Sect. 1.3, we introduced the umbrella term "*process science*" to refer to the broader discipline that combines knowledge from information technology and knowledge from management sciences to improve and run operational processes. Many of the (sub)disciplines mentioned in Fig. 1.6 heavily rely on *modeling* using a variety of formalisms and notations. In this book, we will use transition systems, Petri nets, BPMN, C-nets, EPCs, YAWL, and process trees as example representations. Before providing a "crash course" in these process representations, we briefly reflect on the role of models and the limitations of modeling in process science.

Since the industrial revolution, productivity has been increasing because of technical innovations, improvements in the organization of work, and the use of information technology. Adam Smith (1723–1790) showed the advantages of the division of labor. Frederick Taylor (1856–1915) introduced the initial principles of scientific management. Henry Ford (1863–1947) introduced the production line for the mass production of "black T-Fords". Around 1950 computers and digital communication infrastructures started to influence business processes. This resulted in dramatic changes in the organization of work and enabled new ways of doing business. Today, innovations in computing and communication are still the main drivers behind

W. van der Aalst, *Process Mining*, DOI 10.1007/978-3-662-49851-4_3

change in business processes. So, business processes have become more complex, heavily rely on information systems, and may span multiple organizations. Therefore, process modeling has become of the utmost importance. Process models assist in managing complexity by providing insight and documenting procedures. Information systems need to be configured and driven by precise instructions. Cross-organizational processes can only function properly if there is a common agreement on the required interactions. As a result, process models are widely used in today's organizations.

Operations management, and in particular *operation research*, is a branch of management science heavily relying on modeling. Here a variety of mathematical models ranging from linear programming and project planning to queueing models, Markov chains, and simulation are used. For example, the location of a warehouse is determined using linear programming, server capacity is added on the basis of queueing models, and an optimal route in a container terminal is determined using integer programming. Models are used to reason *about processes* (redesign) and to make decisions *inside processes* (planning and control). The models used in operations management are typically tailored towards a particular analysis technique and only used for answering a specific question. In contrast, process models in BPM typically serve *multiple* purposes. A process model expressed in BPMN may be used to discuss responsibilities, analyze compliance, predict performance using simulation, and configure a WFM system. However, BPM and operations management have in common that making a good model is "an art rather than a science". Creating models is therefore a difficult and error-prone task. Typical errors include:

- *The model describes an idealized version of reality.* When modeling processes the designer tends to concentrate on the "normal" or "desirable" behavior. For example, the model may only cover 80% of the cases assuming that these are representative. Typically this is not the case as the other 20% may cause 80% of the problems. The reasons for such oversimplifications are manifold. The designer and management may not be aware of the many deviations that take place. Moreover, the perception of people may be biased, depending on their role in the organization. Hand-made models tend to be subjective, and often there is a tendency to make things too simple just for the sake of understandability.
- *Inability to adequately capture human behavior.* Although simple mathematical models may suffice to model machines or people working in an assembly line, they are inadequate when modeling people involved in multiple processes and exposed to multiple priorities [139, 163]. A worker who is involved in multiple processes needs to distribute his attention over multiple processes. This makes it difficult to model one process in isolation. Workers also do not work at constant speed. A well-known illustration of this is the so-called *Yerkes–Dodson law* that describes the relation between workload and performance of people [139]. In most processes one can easily observe that people will take more time to complete a task and effectively work fewer hours per day if there is hardly any work to do. Nevertheless, most simulation models sample service times from a fixed probability distribution and use fixed time windows for resource availability.

- *The model is at the wrong abstraction level.* Depending on the input data and the questions that need to be answered, a suitable abstraction level needs to be chosen. The model may be too abstract and thus unable to answer relevant questions. The model may also be too detailed, e.g., the required input cannot be obtained or the model becomes too complex to be fully understood. Consider, for example, a car manufacturer that has a warehouse containing thousands of spare parts. It may be tempting to model all of them in a simulation study to compare different inventory policies. However, if one is not aiming at making statements about a specific spare part, this is not wise. Typically it is very time consuming to change the abstraction level of an existing model. Unfortunately, questions may emerge at different levels of granularity.

These are just some of the problems organizations face when making models by hand. Only experienced designers and analysts can make models that have a good predictive value and can be used as a starting point for a (re)implementation or redesign. An inadequate model can lead to wrong conclusions. Therefore, we advocate the use of event data. Process mining allows for the extraction of models based on *facts*. Moreover, process mining does not aim at creating a single model of the process. Instead, it provides *various views on the same reality at different abstraction levels*. For example, users can decide to look at the most frequent behavior to get a simple model ("80% model"). However, they can also inspect the full behavior by deriving the "100% model" covering all cases observed. Similarly, abstraction levels can be varied to create different views. Process mining can also reveal that people in organizations do not function as "machines". On the one hand, it may be shown that all kinds of inefficiencies take place. On the other hand, process mining can also visualize the remarkable flexibility of some workers to deal with problems and varying workloads.

3.2 Process Models

It is not easy to make good process models. Yet, they are important. Fortunately, process mining can facilitate the construction of better models in less time. Process discovery algorithms like the α-algorithm can automatically generate a process model. As indicated in Chap. 2, various process modeling notations exist. Sometimes the plethora of notations is referred to as the new "tower of Babel". Therefore, we describe only some basic notations. This section does not aim to provide a complete overview of existing process modeling notations. We just introduce the notations that we will use in the remainder. We would like to stress that it is relatively easy to automatically translate process mining results into the desired notation. For example, although the α-algorithm produces a Petri net, it is easy to convert the result into a BPMN model, BPEL model, or UML Activity Diagram. Again we refer to the systematic comparisons in the context of the Workflow Patterns Initiative [155, 191] for details.

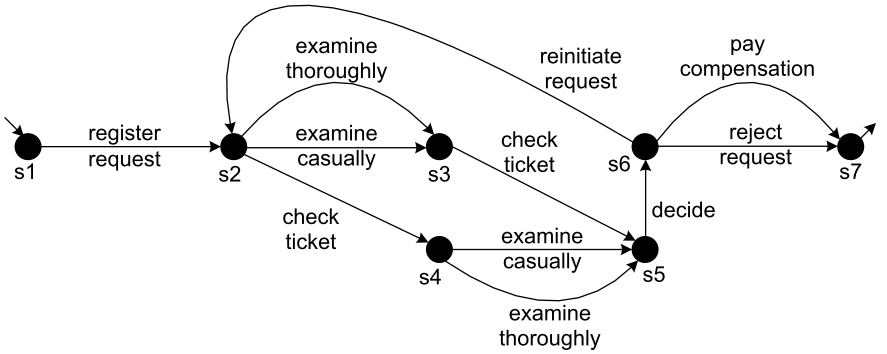


Fig. 3.1 A transition system having one initial state and one final state

In this section we focus on the control-flow perspective of processes. We assume that there is a set of *activity labels* $\mathscr{A}$. The goal of a process model is to decide *which activities* need to be executed and in *what order*. Activities can be executed sequentially, activities can be optional or concurrent, and the repeated execution of the same activity may be possible.

3.2.1 Transition Systems

The most basic process modeling notation is a *transition system*. A transition system consists of *states* and *transitions*. Figure 3.1 shows a transition system consisting of seven states. It models the handling of a request for compensation within an airline as described in Sect. 2.1. The states are represented by black circles. There is one initial state labeled *s1* and one final state labeled *s7*. Each state has a unique label. This label is merely an identifier and has no meaning. Transitions are represented by arcs. Each transition connects two states and is labeled with the name of an activity. Multiple arcs can bear the same label. For example, *check ticket* appears twice.

Definition 3.1 (Transition system) A *transition system* is a triplet $TS = (S, A, T)$ where S is the set of *states*, $A \subseteq \mathscr{A}$ is the set of *activities* (often referred to as *actions*), and $T \subseteq S \times A \times S$ is the set of *transitions*. $S^{start} \subseteq S$ is the set of *initial states* (sometimes referred to as "start" states), and $S^{end} \subseteq S$ is the set of *final states* (sometimes referred to as "accept" states).

The sets S^{start} and S^{end} are defined implicitly. In principle, S can be infinite. However, for most practical applications the state space is finite. In this case the transition system is also referred to as a Finite-State Machine (FSM) or a finite-state automaton.

The transition system depicted in Fig. 3.1 can be formalized as follows: $S = \{s1, s2, s3, s4, s5, s6, s7\}$, $S^{start} = \{s1\}$, $S^{end} = \{s7\}$, $A = \{$*register request*, *examine thoroughly*, *examine casually*, *check ticket*, *decide*, *reinitiate request*, *reject*

request, *pay compensation*}, and $T = \{(s1, \textit{register request}, s2), (s2, \textit{examine casually}, s3), (s2, \textit{examine thoroughly}, s3), (s2, \textit{check ticket}, s4), (s3, \textit{check ticket}, s5), (s4, \textit{examine casually}, s5), (s4, \textit{examine thoroughly}, s5), (s5, \textit{decide}, s6), (s6, \textit{reinitiate request}, s2), (s6, \textit{pay compensation}, s7), (s6, \textit{reject request}, s7)\}$.

Given a transition system one can reason about its behavior. The transition starts in one of the initial states. Any path in the graph starting in such a state corresponds to a possible *execution sequence*. For example, the path *register request*, *examine casually*, *check ticket* in Fig. 3.1 is an example of an execution sequence starting in state $s1$ and ending in $s5$. There are infinitely many execution sequences for this transition system. A path *terminates successfully* if it ends in one of the final states. A path *deadlocks* if it reaches a non-final state without any outgoing transitions. Note that the absence of deadlocks does not guarantee successful termination. The transition system may *livelock*, i.e., some transitions are still enabled but it is impossible to reach one of the final states.

Any process model with executable semantics can be mapped onto a transition system. Therefore, many notions defined for transition systems can easily be translated to higher-level languages such as Petri nets, BPMN, and UML activity diagrams. Consider, for example, the seemingly simple question: "When are two processes the same from a behavioral point of view". As shown in [176], many equivalence notions can be defined. *Trace equivalence* considers two transition systems to be equivalent if their execution sequences are the same. More refined notions like *branching bisimilarity* also take the moment of choice into account. These notions defined for transition systems can be used for any pair of process models as long as the models are expressed in a language with executable semantics (see also Sect. 6.3).

Transition systems are simple but have problems expressing concurrency succinctly. Suppose that there are n parallel activities, i.e., all n activities need to be executed but any order is allowed. There are $n!$ possible execution sequences. The transition system requires 2^n states and $n \times 2^{n-1}$ transitions. This is an example of the well-known "state explosion" problem [135]. Consider for example 10 parallel activities. The number of possible execution sequences is $10! = 3{,}628{,}800$, the number of reachable states is $2^{10} = 1024$, and the number of transitions is $10 \times 2^{10-1} = 5120$. The corresponding Petri net is much more compact and needs only 10 transitions and 10 places to model the 10 parallel activities. Given the concurrent nature of business processes, more expressive models like Petri nets are needed to adequately represent process mining results.

3.2.2 Petri Nets

Petri nets are the oldest and best investigated process modeling language allowing for the modeling of concurrency. Although the graphical notation is intuitive and simple, Petri nets are executable and many analysis techniques can be used to analyze them [82, 117, 149]. In the introduction we already showed an example Petri

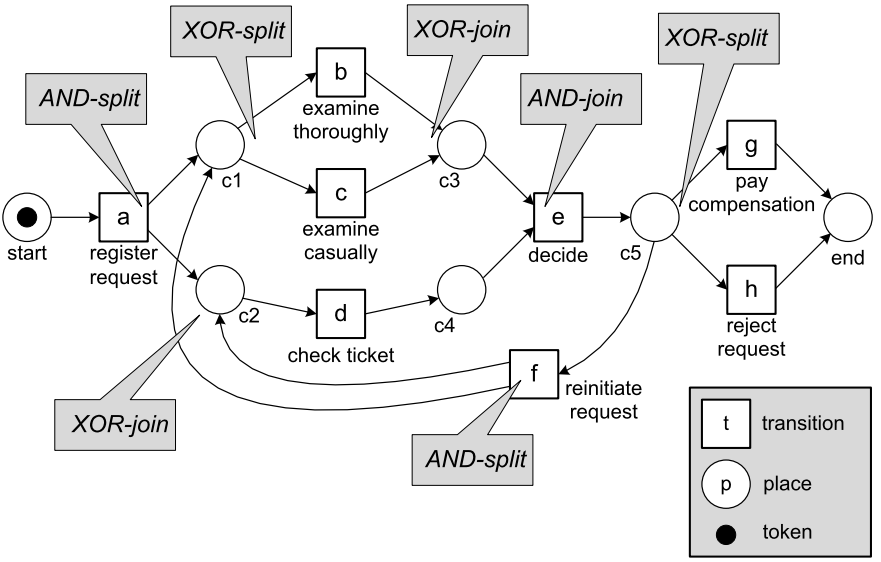


Fig. 3.2 A marked Petri net

net. Figure 3.2 shows the Petri net again with the various constructs highlighted. A Petri net is a bipartite graph consisting of *places* and *transitions*. The network structure is static, but, governed by the firing rule, *tokens* can flow through the network. The state of a Petri net is determined by the distribution of tokens over places and is referred to as its *marking*. In the initial marking shown in Fig. 3.2, there is only one token; *start* is the only marked place.

Definition 3.2 (Petri net) A *Petri net* is a triplet $N = (P, T, F)$ where P is a finite set of *places*, T is a finite set of *transitions* such that $P \cap T = \emptyset$, and $F \subseteq (P \times T) \cup (T \times P)$ is a set of directed arcs, called the *flow relation*. A *marked* Petri net is a pair (N, M), where $N = (P, T, F)$ is a Petri net and where $M \in \mathbb{B}(P)$ is a *multi-set* over P denoting the *marking* of the net. The set of all marked Petri nets is denoted $\mathcal{N}$.

The Petri net shown Fig. 3.2 can be formalized as follows: $P = \{start, c1, c2, c3, c4, c5, end\}$, $T = \{a, b, c, d, e, f, g, h\}$, and $F = \{(start, a), (a, c1), (a, c2), (c1, b), (c1, c), (c2, d), (b, c3), (c, c3), (d, c4), (c3, e), (c4, e), (e, c5), (c5, f), (f, c1), (f, c2), (c5, g), (c5, h), (g, end), (h, end)\}$.

Multi-sets
A marking corresponds to a multi-set of tokens. However, multi-sets are not only used to represent markings; later we will use multi-sets to model event logs where the same trace may appear multiple times. Therefore, we provide some basic notations used in the remainder.

A multi-set (also referred to as *bag*) is like a set in which each element may occur multiple times. For example, $[a, b^2, c^3, d^2, e]$ is the multi-set with nine elements: one a, two b's, three c's, two d's, and one e. The following three multi-set are identical: $[a, b, b, c^3, d, d, e]$, $[e, d^2, c^3, b^2, a]$, and $[a, b^2, c^3, d^2, e]$. Only the number of occurrences of each value matters, not the order. Formally, $\mathbb{B}(D) = D \to \mathbb{N}$ is the set of multi-sets (bags) over a finite domain D, i.e., $X \in \mathbb{B}(D)$ is a multi-set, where for each $d \in D$, $X(d)$ denotes the number of times d is included in the multi-set. For example, if $X = [a, b^2, c^3]$, then $X(b) = 2$ and $X(e) = 0$.

The sum of two multi-sets ($X \uplus Y$), the difference ($X \setminus Y$), the presence of an element in a multi-set ($x \in X$), and the notion of subset ($X \leq Y$) are defined in a straightforward way. For example, $[a, b^2, c^3, d] \uplus [c^3, d, e^2, f^3] = [a, b^2, c^6, d^2, e^2, f^3]$ and $[a, b] \leq [a, b^3, c]$. Moreover, we can also apply these operators to sets, where we assume that a set is a multi-set in which every element occurs exactly once. For example, $[a, b^2] \uplus \{b, c\} = [a, b^3, c]$.

The operators are also robust with respect to the domains of the multi-sets, i.e., even if X and Y are defined on different domains, $X \uplus Y$, $X \setminus Y$, and $X \leq Y$ are defined properly by extending the domain whenever needed.

The marking shown in Fig. 3.2 is $[start]$, i.e., a multi-set containing only one token. The dynamic behavior of such a marked Petri net is defined by the so-called *firing rule*. A transition is *enabled* if each of its input places contains a token. An enabled transition can *fire* thereby consuming one token from each input place and producing one token for each output place. Hence, transition a is enabled at marking $[start]$. Firing a results in the marking $[c1, c2]$. Note that one token is consumed and two tokens are produced. At marking $[c1, c2]$, transition a is no longer enabled. However, transitions b, c, and d have become enabled. From marking $[c1, c2]$, firing b results in marking $[c2, c3]$. Here, d is still enabled, but b and c not anymore. Because of the loop construct involving f there are infinitely many firing sequences starting in $[start]$ and ending in $[end]$.

Assume now that the initial marking is $[start^5]$. Firing a now results in the marking $[start^4, c1, c2]$. At this marking a is still enabled. Firing a again results in marking $[start^3, c1^2, c2^2]$. Transition a can fire five times in a row resulting in marking $[c1^5, c2^5]$. Note that after the first occurrence of a, also b, c, and d are enabled and can fire concurrently.

To formalize the firing rule, we introduce a notation for input (output) places (transitions). Let $N = (P, T, F)$ be a Petri net. Elements of $P \cup T$ are called *nodes*. A node x is an *input node* of another node y if and only if there is a directed arc from x to y (i.e., $(x, y) \in F$). Node x is an *output node* of y if and only if $(y, x) \in F$. For any $x \in P \cup T$, $\bullet x = \{y \mid (y, x) \in F\}$ and $x\bullet = \{y \mid (x, y) \in F\}$. In Fig. 3.2, $\bullet c1 = \{a, f\}$ and $c1\bullet = \{b, c\}$.

Definition 3.3 (Firing rule) Let (N, M) be a marked Petri net with $N = (P, T, F)$ and $M \in \mathbb{B}(P)$. Transition $t \in T$ is *enabled*, denoted $(N, M)[t\rangle$, if and only if

$\bullet t \leq M$. The *firing rule* $\_[\_\rangle\_ \subseteq \mathcal{N} \times T \times \mathcal{N}$ is the smallest relation satisfying for any $(N, M) \in \mathcal{N}$ and any $t \in T$, $(N, M)[t\rangle \Rightarrow (N, M)\ [t\rangle\ (N, (M \setminus \bullet t) \uplus t\bullet)$.

$(N, M)[t\rangle$ denotes that t is enabled at marking M, e.g., $(N, [start])[a\rangle$ in Fig. 3.2. $(N, M)\ [t\rangle\ (N, M')$ denotes that firing this enabled transition results in marking M'. For example, $(N, [start])\ [a\rangle\ (N, [c1, c2])$ and $(N, [c3, c4])\ [e\rangle\ (N, [c5])$.

Let (N, M_0) with $N = (P, T, F)$ be a marked Petri net. A sequence $\sigma \in T^*$ is called a *firing sequence* of (N, M_0) if and only if, for some natural number $n \in \mathbb{N}$, there exist markings $M_1, \ldots, M_n$ and transitions $t_1, \ldots, t_n \in T$ such that $\sigma = \langle t_1 \ldots t_n \rangle$ and, for all i with $0 \leq i < n$, $(N, M_i)[t_{i+1}\rangle$ and $(N, M_i)\ [t_{i+1}\rangle\ (N, M_{i+1})$.[1]

Let (N, M_0) be the marked Petri net shown in Fig. 3.2, i.e., $M_0 = [start]$. The empty sequence $\sigma = \langle\ \rangle$ is enabled in (N, M_0), i.e., $\langle\ \rangle$ is a firing sequence of (N, M_0). The sequence $\sigma = \langle a, b \rangle$ is also enabled and firing σ results in marking $[c2, c3]$. Another possible firing sequence is $\sigma = \langle a, c, d, e, f, b, d, e, g \rangle$. A marking M is *reachable* from the initial marking M_0 if and only if there exists a sequence of enabled transitions whose firing leads from M_0 to M. The set of reachable markings of (N, M_0) is denoted $[N, M_0\rangle$. The marked Petri net shown in Fig. 3.2 has seven reachable markings.

In Fig. 3.2, transitions are identified by a single letter, but also have a longer label describing the corresponding activity. Thus far we ignored these labels.

Definition 3.4 (Labeled Petri net) A *labeled Petri net* is a tuple $N = (P, T, F, A, l)$ where (P, T, F) is a Petri net as defined in Definition 3.2, $A \subseteq \mathscr{A}$ is a set of *activity labels*, and $l \in T \rightarrow A$ is a *labeling function*.

In principle, multiple transitions may bear the same label. One can think of the transition label as the *observable action*. Sometimes one wants to express that particular transitions are not observable. For this we reserve the label τ. A transition t with $l(t) = \tau$ is unobservable. Such transitions are often referred to as *silent* or *invisible*. It is easy to convert any Petri net into a labeled Petri net; just take $A = T$ and $l(t) = t$ for any $t \in T$. The reverse is not always possible, e.g., when several transitions have the same label. It is also possible to convert a marked (labeled) Petri net into a transition system as is shown next.

Definition 3.5 (Reachability graph) Let (N, M_0) with $N = (P, T, F, A, l)$ be a marked labeled Petri net. (N, M_0) defines a transition system $TS = (S, A', T')$ with $S = [N, M_0\rangle$, $S^{start} = \{M_0\}$, $A' = A$, and $T' = \{(M, l(t), M') \in S \times A \times S \mid \exists_{t \in T}\ (N, M)\ [t\rangle\ (N, M')\}$. TS is often referred to as the *reachability graph* of (N, M_0).

Figure 3.3 shows the transition system generated from the labeled marked Petri net shown in Fig. 3.2. States correspond to reachable markings, i.e., multi-sets of

[1] X^* is the set of sequences containing elements of X, i.e., for any $n \in \mathbb{N}$ and $x_1, x_2, \ldots, x_n \in X$: $\langle x_1, x_2, \ldots, x_n \rangle \in X^*$. See also Sect. 5.2.

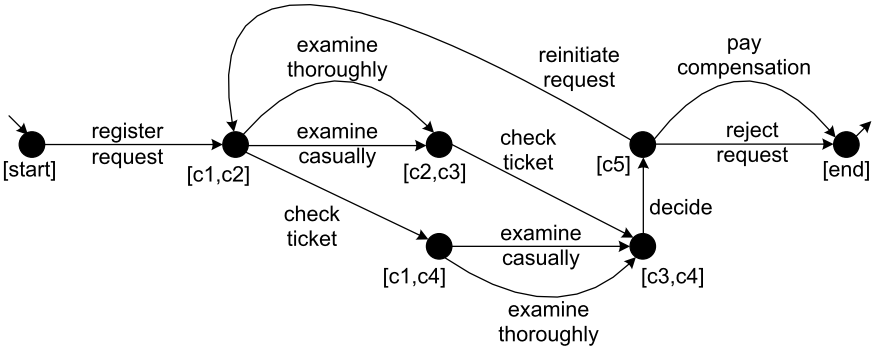


Fig. 3.3 The reachability graph of the marked Petri net shown in Fig. 3.2

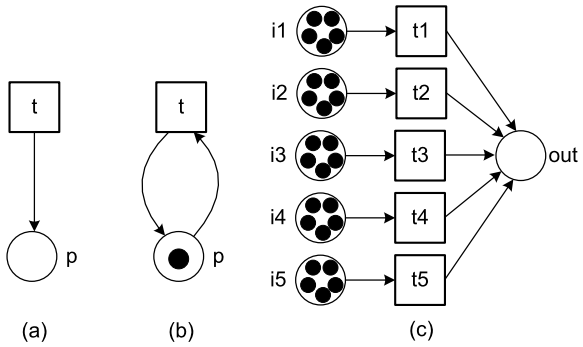


Fig. 3.4 Three Petri nets: (**a**) a Petri net with an infinite state space, (**b**) a Petri net with only one reachable marking, (**c**) a Petri net with 7776 reachable markings

tokens. Note that $S^{start} = \{[start]\}$ is a singleton containing the initial marking of the Petri net. The Petri net does not explicitly define a set of final markings S^{end}. However, in this case it is obvious to take $S^{end} = \{[end]\}$. Later, we will see that it is sometimes useful to distinguish deadlocks and livelocks from successful termination.

Note that we are overloading the term "transition"; the term may refer to a "box" in a Petri net or an "arc" in a transition system. In fact, one transition in a Petri net may correspond to many transitions in the corresponding transition system.

The Petri net in Fig. 3.2 and the transition system in Fig. 3.3 are of similar sizes. If the model contains a lot of concurrency or multiple tokens reside in the same place, then the transition system is much bigger than the Petri net. In fact, a marked Petri net may have infinitely many reachable states. The marked Petri net in Fig. 3.4(a) consists of only one place and one transition. Nevertheless, its corresponding transition system has infinitely many states: $S = \{[p^k] \mid k \in \mathbb{N}\}$. In this example, transition t is continuously enabled because it has no input place. Therefore, it can put any number of tokens in p. The Petri net in Fig. 3.4(b) has two arcs rather than one and now the only reachable state is $[p]$. The marked Petri net in Fig. 3.4(c) shows the effect of concurrency. The corresponding transition system has $6^5 = 7776$ states and 32,400 transitions.

Modern computers can easily compute reachability graphs with millions of states and analyze them. If the reachability graph is infinite, one can resort to the so-called *coverability graph* that presents a kind of over-approximation of the state space [117]. By constructing the reachability graph (if possible) or the coverability graph one can answer a variety of questions regarding the behavior of the process modeled. Moreover, dedicated analysis techniques can also answer particular questions without constructing the state space, e.g., using the linear-algebraic representation of the Petri net. It is outside the scope of this book to elaborate on these. However, we list some generic properties typically investigated in the context of a marked Petri net.

- A marked Petri net (N, M_0) is *k-bounded* if no place ever holds more that k tokens. Formally, for any $p \in P$ and any $M \in [N, M_0\rangle$: $M(p) \leq k$. The marked Petri net in Fig. 3.4(c) is 25-bounded because in none of the 7776 reachable markings there is a place with more than 25 tokens. It is not 24-bounded, because in the final marking place *out* contains 25 tokens.
- A marked Petri net is *safe* if and only if it is 1-bounded. The marked Petri net shown in Fig. 3.2 is safe because in each of the seven reachable markings there is no place holding multiple tokens.
- A marked Petri net is *bounded* if and only if there exists a $k \in \mathbb{N}$ such that it is k-bounded. Figure 3.4(a) shows an unbounded net. The two other marked Petri nets in Fig. 3.4 (i.e., (b) and (c)) are bounded.
- A marked Petri net (N, M_0) is *deadlock free* if at every reachable marking at least one transition is enabled. Formally, for any $M \in [N, M_0\rangle$ there exists a transition $t \in T$ such that $(N, M)[t\rangle$. Figure 3.4(c) shows a net that is not deadlock free because at marking $[out^{25}]$ no transition is enabled. The two other marked Petri nets in Fig. 3.4 are deadlock free.
- A transition $t \in T$ in a marked Petri net (N, M_0) is *live* if from every reachable marking it is possible to enable t. Formally, for any $M \in [N, M_0\rangle$ there exists a marking $M' \in [N, M\rangle$ such that $(N, M')[t\rangle$. A marked Petri net is live if each of its transitions is live. Note that a deadlock-free Petri net does not need to be live. For example, merge the nets (b) and (c) in Fig. 3.4 into one marked Petri net. The resulting net is deadlock free, but not live.

Petri nets have a strong theoretical basis and can capture concurrency well. Moreover, a wide range of powerful analysis techniques and tools exists [117]. Obviously, this succinct model has problems capturing data-related and time-related aspects. Therefore, various types of high-level Petri nets have been proposed. *Colored Petri nets* (CPNs) are the most widely used Petri-net based formalism that can deal with data-related and time-related aspects [82, 149]. Tokens in a CPN carry a data value and have a timestamp. The data value, often referred to as "color", describes the properties of the object modeled by the token. The timestamp indicates the earliest time at which the token may be consumed. Transitions can assign a delay to produced tokens. This way waiting and service times can be modeled. A CPN may be hierarchical, i.e., transitions can be decomposed into subprocesses. This way large models can be structured. CPN Tools is a toolset providing support for the modeling and analysis of CPNs (www.cpntools.org).

3.2.3 Workflow Nets

When modeling business processes in terms of Petri nets, we often consider a subclass of Petri nets known as *WorkFlow nets* (WF-nets) [136, 168]. A WF-net is a Petri net with a dedicated source place where the process starts and a dedicated sink place where the process ends. Moreover, all nodes are on a path from source to sink.

Definition 3.6 (Workflow net) Let $N = (P, T, F, A, l)$ be a (labeled) Petri net and $\bar{t}$ a fresh identifier not in $P \cup T$. N is a *workflow net* (WF-net) if and only if (a) P contains an input place i (also called source place) such that $\bullet i = \emptyset$, (b) P contains an output place o (also called sink place) such that $o\bullet = \emptyset$, and (c) $\bar{N} = (P, T \cup \{\bar{t}\}, F \cup \{(o, \bar{t}), (\bar{t}, i)\}, A \cup \{\tau\}, l \cup \{(\bar{t}, \tau)\})$ is strongly connected, i.e., there is a directed path between any pair of nodes in $\bar{N}$.

$\bar{N}$ is referred to as the short-circuited net [136]. The unique sink place o is connected to the unique source place i in the resulting net.

Figure 3.2 shows an example of a WF-net with $i = start$ and $o = end$. None of the three Petri nets in Fig. 3.4 is a WF-net.

Why are WF-nets particularly relevant for business process modeling? The reason is that the process models used in the context of BPM describe the *life-cycle of cases* of a given kind. Examples of cases are insurance claims, job applications, customer orders, replenishment orders, patients, and credit applications. The process model is instantiated once for each case. Each of these process instances has a well-defined start ("case creation") and end ("case completion"). In-between these points, activities are conducted according to a predefined procedure. One model may be instantiated many times. For example, the process of handling insurance claims may be executed for thousands or even millions of claims. These instances can be seen as copies of the same WF-net, i.e., tokens of different cases are not mixed.

WF-nets are also a natural representation for process mining. There is an obvious relation between the firing sequences of a WF-net and the traces found in event logs. Note that one can only learn models based on examples. In the context of market basket analysis, i.e., finding patterns in what customers buy, one needs many examples of customers buying particular collections of products. Similarly, process discovery uses sequences of activities in which each sequence refers to a particular process instance. These can be seen as firing sequences of an unknown WF-net. Therefore, we will often focus on WF-nets. Recall that the α-algorithm discovered the WF-net in Fig. 2.6 using the set of traces shown in Table 2.2. Every trace corresponds to a case executed from begin to end.

Not every WF-net represents a correct process. For example, a process represented by a WF-net may exhibit errors such as deadlocks, activities that can never become active, livelocks, or garbage being left in the process after termination. Therefore, we define the following well-known correctness criterion [136, 168]:

Definition 3.7 (Soundness) Let $N = (P, T, F, A, l)$ be a WF-net with input place i and output place o. N is *sound* if and only if

- (*safeness*) $(N, [i])$ is safe, i.e., places cannot hold multiple tokens at the same time;
- (*proper completion*) for any marking $M \in [N, [i]\rangle$, $o \in M$ implies $M = [o]$;
- (*option to complete*) for any marking $M \in [N, [i]\rangle$, $[o] \in [N, M\rangle$; and
- (*absence of dead parts*) $(N, [i])$ contains no dead transitions (i.e., for any $t \in T$, there is a firing sequence enabling t).

Note that the option to complete implies proper completion. The WF-net shown in Fig. 3.2 is sound. Soundness can be verified using standard Petri-net-based analysis techniques. In fact soundness corresponds to liveness and safeness of the corresponding short-circuited net $\bar{N}$ introduced in Definition 3.6 [136]. This way efficient algorithms and tools can be applied. An example of a tool tailored towards the analysis of WF-nets is *Woflan* [179]. This functionality is also embedded in our process mining tool *ProM* described in Sect. 11.3.

3.2.4 YAWL

YAWL is both a workflow modeling *language* and an open-source workflow *system* [132]. The acronym YAWL stands for "Yet Another Workflow Language". The development of the YAWL language was heavily influenced by the *Workflow Patterns Initiative* [155, 191] mentioned earlier. Based on a systematic analysis of the constructs used by existing process modeling notations and workflow languages, a large collection of patterns was identified. These patterns cover all workflow perspectives, i.e., there are control-flow patterns, data patterns, resource patterns, change patterns, exception patterns, etc. The aim of YAWL is to offer direct support for many patterns while keeping the language simple. It can be seen as a reference implementation of the most important workflow patterns. Over time, the YAWL language and the YAWL system have increasingly become synonymous and have garnered widespread interest from both practitioners and the academic community alike. YAWL is currently one of the most widely used open-source workflow systems.

Here we restrict ourselves to the control-flow perspective. Figure 3.5 shows the main constructs. Each process has a dedicated start and end condition, like in WF-nets. Activities in YAWL are called *tasks*. *Conditions* in YAWL correspond to places in Petri nets. However, it is also possible to directly connect tasks without putting a condition in-between. Tasks have—depending on their type—a well-defined split and join semantics. An *AND-join/AND-split* task behaves like a transition, i.e., it needs to consume one token via each of the incoming arcs and produces a token along each of the outgoing arcs. An *XOR-split* selects precisely one of its outgoing arcs. The selection is based on evaluating data conditions. Only one token is produced and sent along the selected arc. An *XOR-join* is enabled once for every incoming token and does not need to synchronize. An *OR-split* selects one or more of its outgoing arcs. This selection is again based on evaluating data conditions. Note

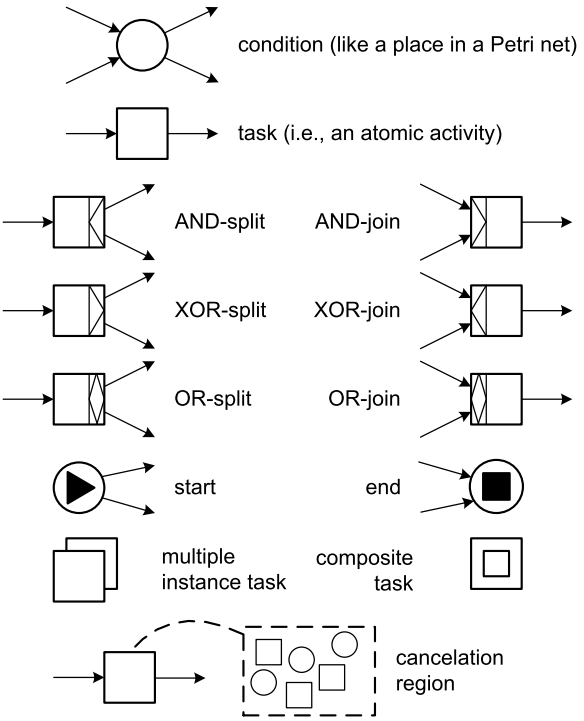


Fig. 3.5 YAWL notation

that an OR-split may select 2 out of three 3 outgoing arcs. The semantics of the *OR-join* are more involved. The OR-join requires at least one input token, but also synchronizes tokens that are "on their way" to the OR-join. As long as another token may arrive via one of the ingoing arcs, the OR-join waits. YAWL also supports *cancelation regions*. A task may have a cancelation region consisting of conditions, tasks, and arcs. Once the task completes all tokens are removed from this region. Note that tokens for the task's output conditions are produced after emptying the cancelation region. YAWL's cancelation regions provide a powerful mechanism to abort work in parallel branches and to reset parts of the workflow. Tasks in a YAWL model can be *atomic* or *composite*. A composite task refers to another YAWL model. This way models can be structured hierarchically. Atomic and composite tasks can be instantiated multiple times in parallel. For example, when handling a customer order, some tasks needs to be executed for every order line. These order lines can be processed in any order. Therefore, a loop construct is less suitable. Figure 3.5 shows the icon for such a multiple instance task and all other constructs just mentioned.

Figure 3.6 shows an example YAWL model for the handling of a request for compensation within an airline. To show some of the features of YAWL, we extended the process described in Sect. 2.1 with some more complex behaviors. In the new model it is possible that both examinations are executed. By using an OR-split and an OR-join *examine causally* and/or *examine thoroughly* are executed. The model has also been extended with a cancelation region (see dotted box in Fig. 3.6). As

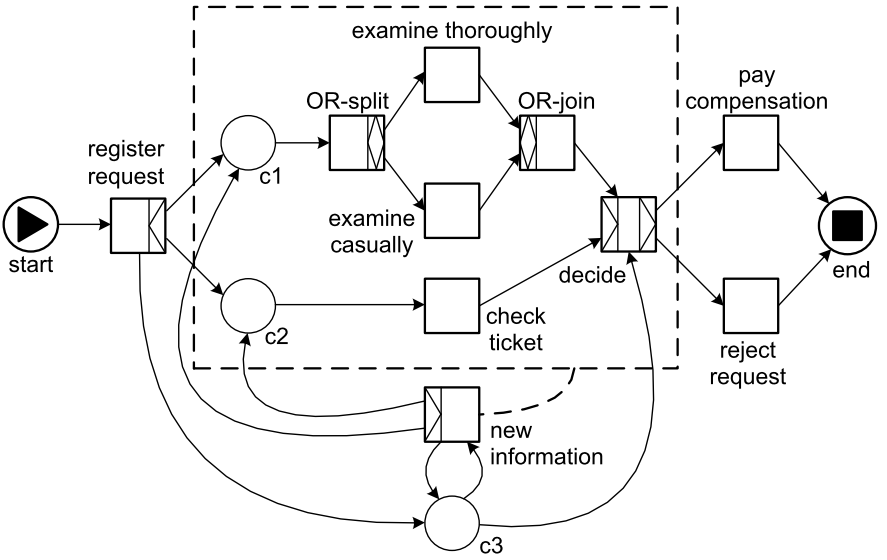


Fig. 3.6 Process model using the YAWL notation

long as there is a token in *c3*, task *new information* may be executed. When this happens, all tokens are removed from the region, i.e., checks and examinations are aborted. Task *new information* does not need to know where all tokens are and after the reset by this task the new state is $[c1, c2, c3]$. Explicit choices in YAWL (i.e., XOR/OR-splits) are driven by data conditions. In the Petri net in Fig. 3.2, all choices were non-deterministic. In the example YAWL model, the decision may be derived from the outcome of the check and the examination(s), i.e., the XOR-split *decide* may be based on data created in earlier tasks. As indicated, both the YAWL language and the YAWL system cover all relevant perspectives (resources, data, exceptions, etc.). For example, it is possible to model that decisions are taken by the manager and that it is not allowed that two examinations for the same request are done by the same person (4-eyes principle) [132].

3.2.5 Business Process Modeling Notation (BPMN)

Recently, the *Business Process Modeling Notation* (BPMN) has become one of the most widely used languages to model business processes. BPMN is supported by many tool vendors and has been standardized by the OMG [110]. Figure 3.7 shows the BPMN model already introduced in Sect. 2.1.

Figure 3.8 shows a small subset of all notational elements. Atomic activities are called *tasks*. Like in YAWL activities can be nested. Most of the constructs can be easily understood after the introduction to YAWL. A notable difference is that the

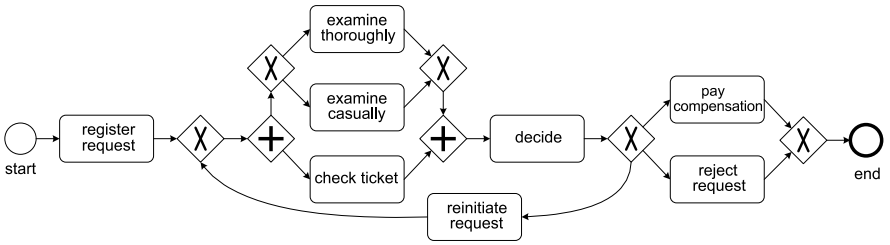
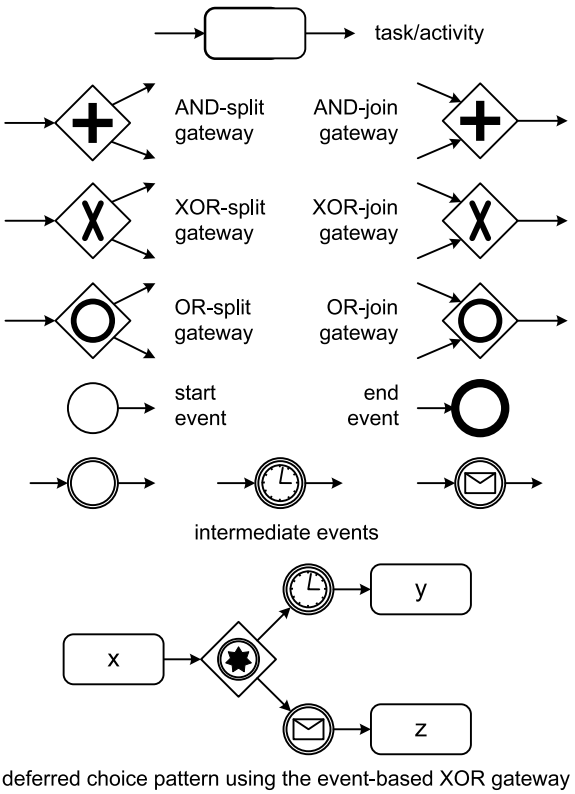


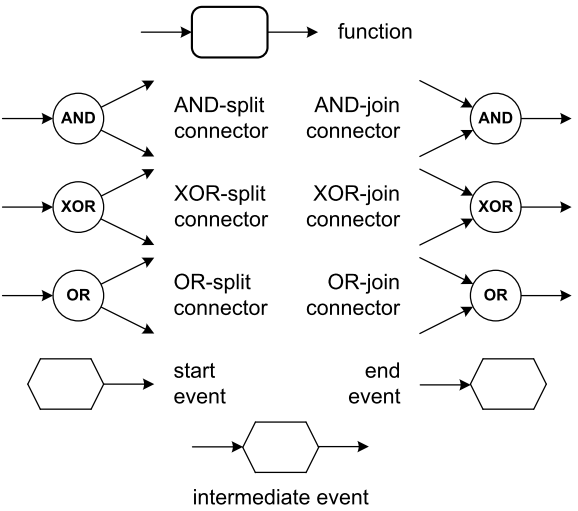
Fig. 3.7 Process model using the BPMN notation

Fig. 3.8 BPMN notation



routing logic is not associated with tasks but with separate *gateways*. Figure 3.8 shows that there are split and join gateways of different types: AND, XOR, OR. The splits are based on data conditions. An *event* is comparable to a place in a Petri net. However, the semantics of places in Petri nets and events in BPMN are quite different. There is no need to insert events in-between activities and events cannot have multiple input or output arcs. *Start events* have one outgoing arc, *intermediate events* have one incoming and one outgoing arc, and *end events* have one incoming arc. Unlike in YAWL or a Petri net, one cannot have events with multiple

Fig. 3.9 EPC notation



incoming or outgoing arcs; splitting and joining needs to be done using gateways. To model the so-called *deferred choice* workflow pattern [155] one needs to use the event-based XOR gateway shown in Fig. 3.8. This illustrates the use of events. After executing task x there is a race between two events. One of the events is triggered by a timeout. The other event is triggered by an external massage. The first event to occur determines the route taken. If the message arrives before the timer goes off, task z is executed. If the timer goes off before the massage arrives, task y is executed. Note that this construct can easily be modeled in YAWL using a condition with two output arcs.

Figure 3.8 shows just a tiny subset of all notations provided by BPMN. Most vendors support only a small subset of BPMN in their products. Moreover, users typically use only few BPMN constructs. In [193], it was shown that the average subset of BPMN used in real-life models consists of less than 10 different symbols (despite the more than 50 distinct graphical elements offered to the modeler). For this reason, we will be rather pragmatic when it comes to process models and their notation.

3.2.6 Event-Driven Process Chains (EPCs)

Event-driven Process Chains (EPCs) provide a classical notation to model business processes [126]. The notation is supported by products such as ARIS and SAP R/3. Basically, EPCs cover a limited subset of BPMN and YAWL while using a dedicated graphical notation.

Figure 3.9 provides an overview of the different notational elements. *Functions* correspond to activities. A function has precisely one input arc and one output arc. Therefore, splitting and joining can only be modeled using *connectors*. These are

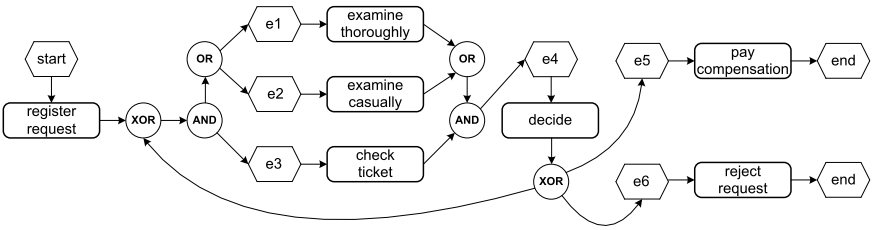


Fig. 3.10 Process model using the EPC notation

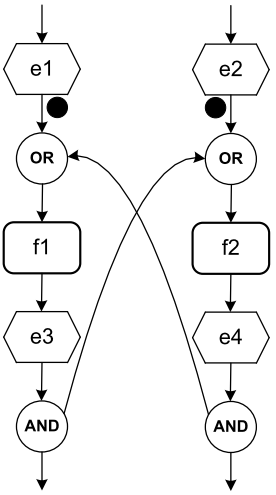


Fig. 3.11 The so-called "vicious circle" expressed using the EPC notation

comparable to the gateways in BPMN. Again splits and joins of type AND, XOR, and OR are supported. Like in BPMN there are three types of *events* (start, intermediate, and end). Events and functions need to alternate along any path, i.e., it is not allowed to connect events to events or functions to functions.

Figure 3.10 shows another variation of the process for handling a request for compensation. Note that, because of the two OR connectors, it is possible to do both examinations or just one.

The EPC notation was one of the first notations allowing for OR splits and joins. However, the people who developed and evangelized EPCs did not provide clear semantics nor some reference implementation [154]. This triggered lively debates resulting in various proposals and alternative implementations. Consider, for example, the so-called "vicious circle" shown in Fig. 3.11. The two tokens show the state of this process fragment; events *e1* and *e2* hold a token. It is unclear what could happen next, because both OR-joins depend on one another.

Should the OR-join below *e1* block or not? Suppose that this OR-join blocks, then by symmetry also the other OR-join following *e2* should block and the whole EPC deadlocks in the state shown Fig. 3.11. This seems to be wrong because if it deadlocks, the OR join will never receive an additional token and hence should

not have waited in the first place. Suppose that the OR-join following *e1* does not block. By symmetry the other OR-join should also not block and both *f1* and *f2* are executed and tokens flow towards both OR-joins via the two AND-splits. However, this implies that the OR-joins should both have blocked. Hence, there is a *paradox* because all possible decisions are wrong.

The vicious circle paradox shows that higher-level constructs may introduce all kinds of subtle semantic problems. Despite these problems and the different notations, the core concepts of the various languages are very similar.

3.2.7 Causal Nets

The notations discussed thus far connect activities (i.e., transitions, tasks, functions) through model elements like places (Petri nets), conditions (YAWL), connectors and events (EPC), gateways and events (BPMN). These elements interconnect activities but do not leave any "marks" in the event log, i.e., they need to be inferred by analyzing the behavior. Since the log does not provide concrete information about places, conditions, connectors, gateways and events, some mining algorithms use a representation consisting of just activities and no connecting elements [4, 12, 66, 183, 184].

Causal nets are a representation tailored towards process mining. A causal net is a graph where nodes represent activities and arcs represent causal dependencies. Each activity has a set of possible *input bindings* and a set of possible *output bindings*. Consider, for example, the causal net shown in Fig. 3.12. Activity a has only an empty input binding as this is the start activity. There are two possible output bindings: $\{b, d\}$ and $\{c, d\}$. This means that a is followed by either b and d, or c and d. Activity e has two possible input bindings ($\{b, d\}$ and $\{c, d\}$) and three possible output bindings ($\{g\}$, $\{h\}$, and $\{f\}$). Hence, e is preceded by either b and d, or c and d, and is succeeded by just g, h or f. Activity z is the end activity having two input bindings and one output binding (the empty binding). This activity has been added to create a unique end point. All executions commence with start activity a and finish with end activity z. As will be shown later, the causal net shown in Fig. 3.12 and the Petri net shown in Fig. 3.2 are trace equivalent, i.e., they both allow for the same set of traces. However, there are no places in the causal net; the routing logic is solely represented by the possible input and output bindings.

Definition 3.8 (Causal net) A *Causal net* (C-net) is a tuple $C = (A, a_i, a_o, D, I, O)$ where:

- $A \subseteq \mathscr{A}$ is a finite set of *activities*;
- $a_i \in A$ is the *start activity*;
- $a_o \in A$ is the *end activity*;
- $D \subseteq A \times A$ is the *dependency relation*,
- $AS = \{X \subseteq \mathscr{P}(A) \mid X = \{\emptyset\} \ \vee \ \emptyset \notin X\}$;[2]

[2] $\mathscr{P}(A) = \{A' \mid A' \subseteq A\}$ is the powerset of A. Hence, elements of AS are *sets of sets* of activities.

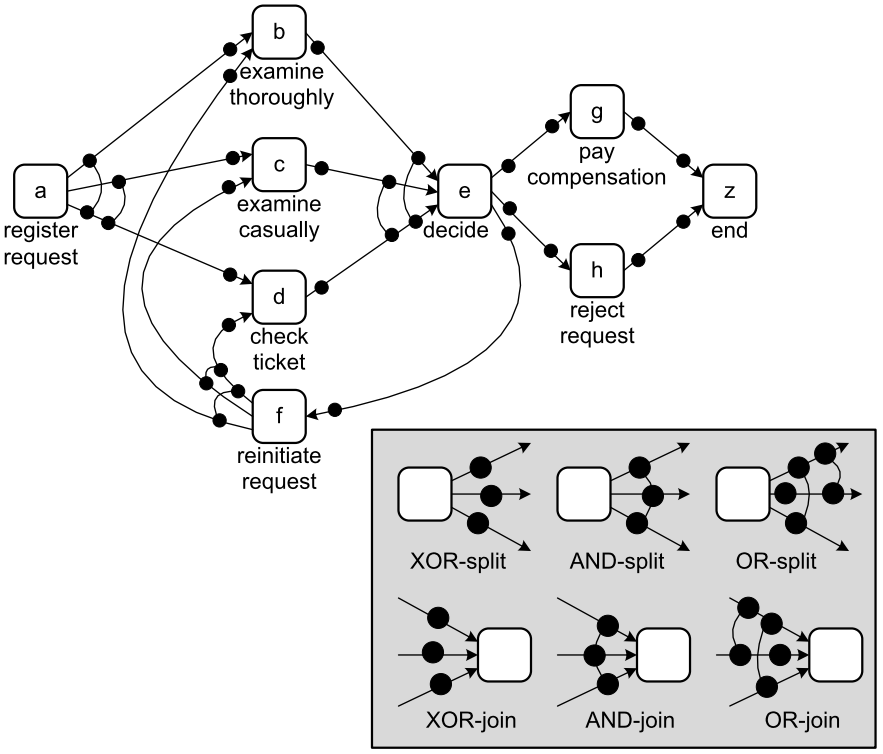


Fig. 3.12 Causal net C_1

- $I \in A \rightarrow AS$ defines the set of possible *input bindings* per activity; and
- $O \in A \rightarrow AS$ defines the set of possible *output bindings* per activity,

such that

- $D = \{(a_1, a_2) \in A \times A \mid a_1 \in \bigcup_{as \in I(a_2)} as\}$;
- $D = \{(a_1, a_2) \in A \times A \mid a_2 \in \bigcup_{as \in O(a_1)} as\}$;
- $\{a_i\} = \{a \in A \mid I(a) = \{\emptyset\}\}$;
- $\{a_o\} = \{a \in A \mid O(a) = \{\emptyset\}\}$; and
- all activities in the graph (A, D) are on a path from a_i to a_o.

The C-net of Fig. 3.12 can be described as follows. $A = \{a, b, c, d, e, f, g, h, z\}$ is the set of activities, $a = a_i$ is the unique start activity, and $z = a_o$ is the unique end activity. The arcs shown in Fig. 3.12 visualize the dependency relation $D = \{(a, b), (a, c), (a, d), (b, e), \ldots, (g, z), (h, z)\}$. Functions I and O describe the sets of possible input and output bindings. $I(a) = \{\emptyset\}$ is the set of possible input bindings of a, i.e., the only input binding is the empty set of activities. $O(a) = \{\{b, d\}, \{c, d\}\}$ is the set of possible output bindings of a, i.e., activity a is followed by d and either b or c. $I(b) = \{\{a\}, \{f\}\}$, $O(b) = \{\{e\}\}$, $\ldots$,

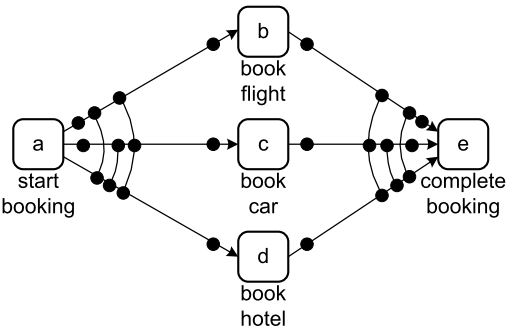


Fig. 3.13 Causal net C_2

$I(z) = \{\{g\}, \{h\}\}$, $O(z) = \{\emptyset\}$. Note that any element of AS is a set of sets of activities, e.g., $\{\{b, d\}, \{c, d\}\} \in AS$. If one of the elements is the empty set, then there cannot be any other elements, i.e., for any $X \in AS$: $X = \{\emptyset\}$ or $\emptyset \notin X$. This implies that only the unique start activity a_i has the empty binding as (only) possible input binding. Similarly, only the unique end activity a_o has the empty binding as (only) possible output binding.

An *activity binding* is a tuple (a, as^I, as^O) denoting the occurrence of activity a with input binding as^I and output binding as^O. For example, $(e, \{b, d\}, \{f\})$ denotes the occurrence of activity e in Fig. 3.12 while being preceded by b and d, and succeeded by f.

Definition 3.9 (Binding) Let $C = (A, a_i, a_o, D, I, O)$ be a C-net. $B = \{(a, as^I, as^O) \in A \times \mathscr{P}(A) \times \mathscr{P}(A) \mid as^I \in I(a) \wedge as^O \in O(a)\}$ is the set of *activity bindings*. A *binding sequence* σ is a sequence of activity bindings, i.e., $\sigma \in B^*$.

A possible binding sequence for the C-net of Fig. 3.12 is $\langle (a, \emptyset, \{b, d\}), (b, \{a\}, \{e\}), (d, \{a\}, \{e\}), (e, \{b, d\}, \{g\}), (g, \{e\}, \{z\}), (z, \{g\}, \emptyset)\rangle$.

Figure 3.13 shows another C-net modeling the booking of a trip. After activity a (*start booking*) there are three possible activities: b (*book flight*), c (*book car*), and d (*book hotel*). The process ends with activity e (*complete booking*). $O(a) = I(e) = \{\{b\}, \{c\}, \{b, d\}, \{c, d\}, \{b, c, d\}\}$, $I(a) = O(e) = \{\emptyset\}$, $I(b) = I(c) = I(d) = \{\{a\}\}$, and $O(b) = O(c) = O(d) = \{\{e\}\}$. A possible binding sequence for the C-net of Fig. 3.12 is $\langle (a, \emptyset, \{b, d\}), (d, \{a\}, \{e\}), (b, \{a\}, \{e\}), (e, \{b, d\}, \emptyset)\rangle$, i.e., the scenario in which a flight and a hotel are booked. Note that Fig. 3.13 does not allow for booking just a hotel nor is it possible to just book a flight and a car.

A binding sequence is *valid* if a predecessor activity and successor activity always "agree" on their bindings. For a predecessor activity x and successor activity y we need to see the following "pattern": $\langle \ldots, (x, \{\ldots\}, \{y, \ldots\}), \ldots, (y, \{x, \ldots\}, \{\ldots\}), \ldots\rangle$, i.e., the occurrence of activity x with y in its output binding needs to be followed by the occurrence of activity y and the occurrence of activity y with x in its input binding needs to be preceded by the occurrence of activity x. To formalize the notion of a valid sequence, we first define the notion of *state*.

Definition 3.10 (State) Let $C = (A, a_i, a_o, D, I, O)$ be a C-net. $S = \mathbb{B}(A \times A)$ is the *state space* of C. $s \in S$ is a *state*, i.e., a multi-set of pending *obligations*. Function $\psi \in B^* \rightarrow S$ is defined inductively: $\psi(\langle\ \rangle) = [\ ]$ and $\psi(\sigma \oplus (a, as^I, as^O)) = (\psi(\sigma) \setminus (as^I \times \{a\})) \uplus (\{a\} \times as^O)$ for any binding sequence $\sigma \oplus (a, as^I, as^O) \in B^*$.[3] $\psi(\sigma)$ is the state after executing binding sequence σ.

Consider C-net C_1 shown in Fig. 3.12. Initially there are no pending "obligations", i.e., no output bindings have been enacted without having corresponding input bindings. If activity binding $(a, \emptyset, \{b, d\})$ occurs, then $\psi(\langle(a, \emptyset, \{b, d\})\rangle) = \psi(\langle\ \rangle) \setminus (\emptyset \times \{a\}) \uplus (\{a\} \times \{b, d\}) = [\ ] \setminus [\ ] \uplus [(a, b), (a, d)] = [(a, b), (a, d)]$. State $[(a, b), (a, d)]$ denotes the obligation to execute both b and d using input bindings involving a. Input bindings remove pending obligations whereas output bindings create new obligations.

A *valid sequence* is a binding sequence that (a) starts with start activity a_i, (b) ends with end activity a_o, (c) only removes obligations that are pending, and (d) ends without any pending obligations. Consider, for example, the valid sequence $\sigma = \langle(a, \emptyset, \{b, d\}), (d, \{a\}, \{e\}), (b, \{a\}, \{e\}), (e, \{b, d\}, \emptyset)\rangle$ for C-net C_2 in Fig. 3.13:

$$\psi(\langle\ \rangle) = [\ ]$$
$$\psi(\langle(a, \emptyset, \{b, d\})\rangle) = [(a, b), (a, d)]$$
$$\psi(\langle(a, \emptyset, \{b, d\}), (d, \{a\}, \{e\})\rangle) = [(a, b), (d, e)]$$
$$\psi(\langle(a, \emptyset, \{b, d\}), (d, \{a\}, \{e\}), (b, \{a\}, \{e\})\rangle) = [(b, e), (d, e)]$$
$$\psi(\langle(a, \emptyset, \{b, d\}), (d, \{a\}, \{e\}), (b, \{a\}, \{e\}), (e, \{b, d\}, \emptyset)\rangle) = [\ ]$$

Sequence σ indeed starts with start activity a, ends with end activity e, only removes obligations that are pending (i.e., for every input binding there was an earlier output binding), and ends without any pending obligations: $\psi(\sigma) = [\ ]$.

Definition 3.11 (Valid) Let $C = (A, a_i, a_o, D, I, O)$ be a C-net and $\sigma = \langle(a_1, as_1^I, as_1^O), (a_2, as_2^I, as_2^O), \ldots, (a_n, as_n^I, as_n^O)\rangle \in B^*$ a binding sequence. σ is a *valid sequence* of C if and only if:

- $a_1 = a_i$, $a_n = a_o$, and $a_k \in A \setminus \{a_i, a_o\}$ for $1 < k < n$;
- $\psi(\sigma) = [\ ]$; and
- for any prefix $\langle(a_1, as_1^I, as_1^O), (a_2, as_2^I, as_2^O), \ldots, (a_k, as_k^I, as_k^O)\rangle = \sigma' \oplus (a_k, as_k^I, as_k^O) \in pref(\sigma)$: $(as_k^I \times \{a_k\}) \leq \psi(\sigma')$.

$V(C)$ is the set of all valid sequences of C.

[3] $\sigma_1 \oplus \sigma_2$ is the concatenation of two sequences, e.g., $\langle a, b, c\rangle \oplus \langle d, e\rangle = \langle a, b, c, d, e\rangle$. It is also possible to concatenate a sequence and an element, e.g., $\langle a, b, c\rangle \oplus d = \langle a, b, c, d\rangle$. Recall that X^* is the set of all sequences containing elements of X and $\langle\ \rangle$ is the empty sequence. See also Sect. 5.2 for more notations for sequences.

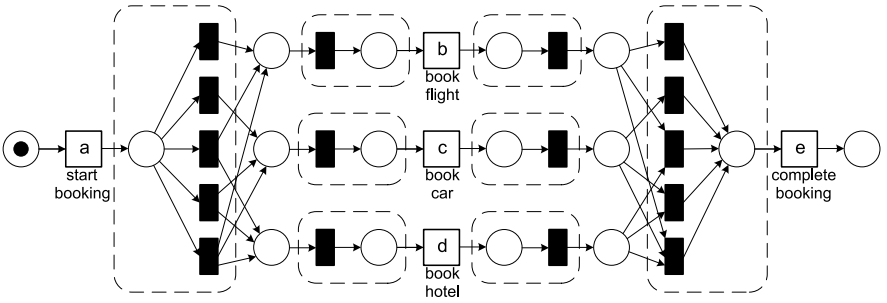
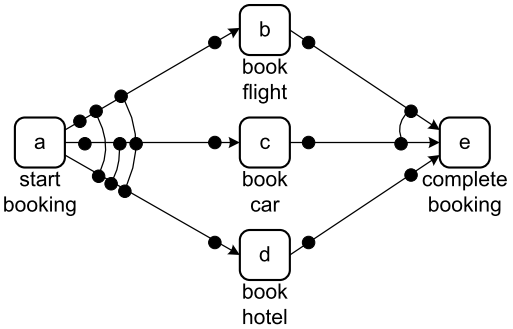


Fig. 3.14 A C-net transformed into a WF-net with silent transitions: every "sound run" of the WF-net corresponds to a valid sequence of the C-net C_2 shown in Fig. 3.13

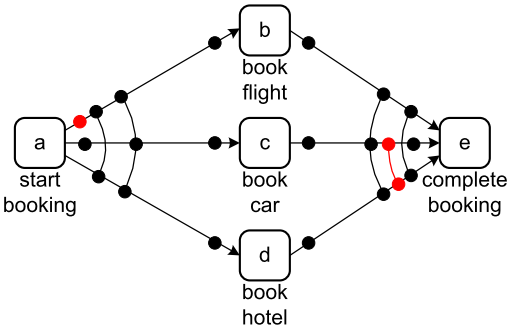
The first requirement states that valid sequences start with a_i and end with a_o (a_i and a_o cannot appear in the middle of valid sequence). The second requirement states that at the end there should not be any pending obligations. (One can think of this as the constraint that no tokens left in the net.) The last requirement considers all non-empty prefixes of $\sigma, \langle (a_1, as_1^I, as_1^O), (a_2, as_2^I, as_2^O), \ldots, (a_k, as_k^I, as_k^O)\rangle$. The last activity binding of the prefix (i.e., (a_k, as_k^I, as_k^O)) should only remove pending obligations, i.e., $(as_k^I \times \{a_k\}) \leq \psi(\sigma')$ where $as_k^I \times \{a_k\}$ are the obligations to be removed and $\psi(\sigma')$ are the pending obligations just before the occurrence of the k-th binding. (One can think of this as the constraint that one cannot consume tokens that have not been produced.)

Figure 3.13 has 12 valid sequences: only b is executed ($\langle (a, \emptyset, \{b\}), (b, \{a\}, \{e\}), (e, \{b\}, \emptyset)\rangle$), only c is executed (besides a and e), b and d are executed (two possibilities), c and d are executed (two possibilities), and b, c and d are executed ($3! = 6$ possibilities). The C-net in Fig. 3.12 has infinitely many valid sequences because of the loop construct involving f. For example, $\langle (a, \emptyset, \{c, d\}), (c, \{a\}, \{e\}), (d, \{a\}, \{e\}), (e, \{c, d\}, \{f\}), (f, \{e\}, \{c, d\}), (c, \{f\}, \{e\}), (d, \{f\}, \{e\}), (e, \{c, d\}, \{g\}), (g, \{e\}, \{z\}), (z, \{g\}, \emptyset)\rangle$.

For the semantics of a C-net we only consider valid sequences, i.e., *invalid sequences are not part of the behavior* described by the C-net. This means that C-nets do not use plain "token-game like semantics" as in BPMN, Petri nets, EPCs, and YAWL. The semantics of C-nets are more declarative as they are defined over complete sequences rather than a local firing rule. This is illustrated by the WF-net shown in Fig. 3.14. This WF-net aims to model the semantics of the C-net C_2 in Fig. 3.13. The input and output bindings are modeled by *silent transitions*. In Fig. 3.14, these are denoted by black rectangles without labels. Note that the WF-net also allows for many invalid sequences. For example, it is possible to enable b, c and d. After firing b it is possible to fire e without firing c and d. This firing sequence does not correspond to a valid sequence because there are still pending commitments when executing the end activity e. However, if we only consider firing sequences of the WF-net that start with a token in the source place and end with a token in the sink place, then these match one-to-one with the valid sequences in $V(C_2)$.



(a) unsound because there are no valid sequences



(b) unsound although there exist valid sequences

Fig. 3.15 Two C-nets that are not sound. The first net does not allow for any valid sequence, i.e., $V(C) = \emptyset$. The second net has valid sequences but also shows input/output bindings that are not realizable

The C-net shown in Fig. 3.12 and the WF-net shown in Fig. 3.2 are trace equivalent. Recall that in this comparison we consider all possible firing sequences of the WF-net and only valid sequences for the C-net.

We defined the notion of soundness for WF-nets (Definition 3.7) to avoid process models that have deadlocks, livelocks, and other anomalies. A similar notion can be defined for C-nets.

Definition 3.12 (Soundness of C-nets) A C-net $C = (A, a_i, a_o, D, I, O)$ is *sound* if (a) for all $a \in A$ and $as^I \in I(a)$ there exists a $\sigma \in V(C)$ and $as^O \subseteq A$ such that $(a, as^I, as^O) \in \sigma$, and (b) for all $a \in A$ and $as^O \in O(a)$ there exists a $\sigma \in V(C)$ and $as^I \subseteq A$ such that $(a, as^I, as^O) \in \sigma$.

Since the semantics of C-nets already enforce "proper completion" and the "option to complete", we only need to make sure that there are valid sequences and that all parts of the C-net can potentially be activated by such a valid sequence. The C-nets C_1 and C_2 in Figs. 3.12 and 3.13 are sound. Figure 3.15 shows two C-nets that are not sound. In Fig. 3.15(a), there are no valid sequences because the output bindings of a and the input bindings of e do not match. For example, consider the binding sequence $\sigma = \langle (a, \emptyset, \{b\}), (b, \{a\}, \{e\}) \rangle$. Sequence σ cannot be extended into a valid sequence because $\psi(\sigma) = [(b, e)]$ and $\{b\} \notin I(e)$, i.e., the input bind-

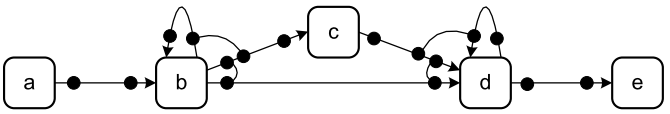


Fig. 3.16 A sound C-net that has no corresponding WF-net

ings of e do not allow for just booking a flight whereas the output bindings of a do. In Fig. 3.15(b), there are valid sequences, e.g., $\langle(a, \emptyset, \{c\}), (c, \{a\}, \{e\}), (e, \{c\}, \emptyset)\rangle$. However, not all bindings appear in one or more valid sequences. For example, the output binding $\{b\} \in O(a)$ does not appear in any valid sequence, i.e., after selecting just a flight the sequence cannot be completed properly. The input binding $\{c, d\} \in I(e)$ also does not appear in any valid sequence, i.e., the C-net suggests that only a car and hotel can be booked but there is no corresponding valid sequence.

Figure 3.16 shows an example of a sound C-net. One of the valid binding sequences for this C-net is $\langle(a, \emptyset, \{b\}), (b, \{a\}, \{b, c\}), (b, \{b\}, \{c, d\}), (c, \{b\}, \{d\}), (c, \{b\}, \{d\}), (d, \{b, c\}, \{d\}), (d, \{c, d\}, \{e\}), (e, \{d\}, \emptyset)\rangle$, i.e., the sequence $\langle a, b, b, c, c, d, d, e\rangle$. This sequence covers all the bindings. Therefore, the C-net is sound. Examples of other valid sequences are $\langle a, b, c, d, e\rangle$, $\langle a, b, c, b, c, d, d, e\rangle$, and $\langle a, b, b, b, c, c, c, d, d, d, e\rangle$. Figure 3.16 illustrates the expressiveness of C-nets. Note that there is no sound WF-net that reproduces exactly the set of valid sequences of this C-net. If we use the construction shown in Fig. 3.14 for the C-net of Fig. 3.16, we get a WF-net that is able to simulate the valid sequences. However, the resulting WF-net also allows for invalid behavior and it is impossible modify the model such that the set of firing sequences coincides with the set of valid sequences.

Causal nets are particularly suitable for process mining given their declarative nature and expressiveness without introducing all kinds of additional model elements (places, conditions, events, gateways, etc.). Several process discovery and conformance checking approaches use a similar representation [4, 12, 66, 183, 184]. In Chap. 7, we elaborate on this when discussing some of the more advanced process mining algorithms.

3.2.8 Process Trees

Petri nets, WF-nets, BPMN models, EPCs, YAWL models, and UML activity diagrams may suffer from deadlocks, livelocks, and other anomalies. Models having undesirable properties *independent* of the event log are called *unsound*. One does not need to look at the event log to see that an unsound model cannot describe the observed behavior well. Process discovery approaches using any of the graph-based process notations mentioned may produce unsound models. In fact, the majority of models in the search space tend to be unsound. This complicates discovery. C-nets address this problem by using more relaxed semantics. It is also possible to use

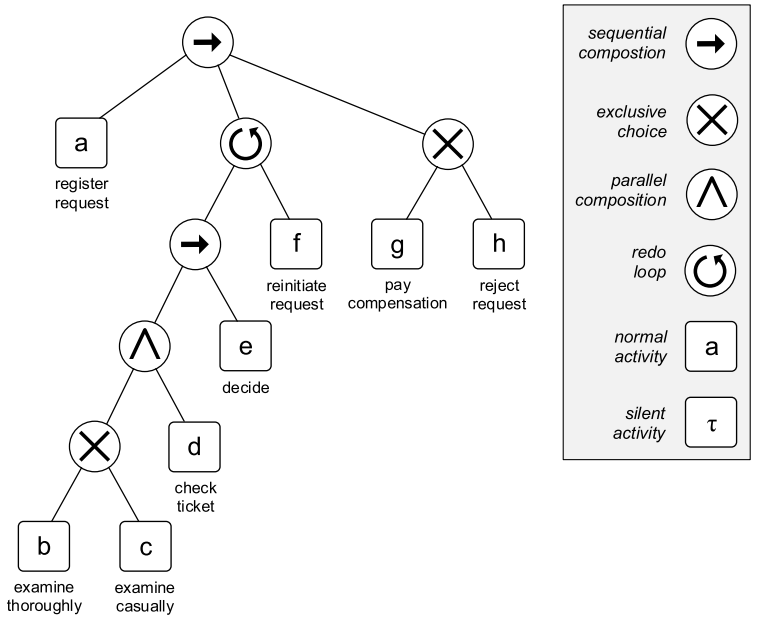


Fig. 3.17 Process tree $\rightarrow(a, \circlearrowleft(\rightarrow(\wedge(\times(b, c), d), e), f), \times(g, h))$ showing the different process tree operators

block-structured models that are sound by construction. In this section, we introduce *process trees* as a notation to represent such block-structured models. A process tree is a hierarchical process model where the (inner) nodes are operators such as sequence and choice and the leaves are activities.

Process trees are tailored towards process discovery. A range of *inductive process discovery* techniques exists for process trees [88–91]. These techniques benefit from the fact that the representation ensures soundness. The family of inductive mining techniques has variants that can handle infrequent behavior and deal with huge models and logs while ensuring formal correctness criteria such as the ability to rediscover the original model (see Sect. 7.5). Also the ETM (Evolutionary Tree Miner) approach described in [26] exploits the process tree representation. The fact that the search space is limited to sound models is a key ingredient of this highly flexible genetic process mining approach.

Figure 3.17 shows a *process tree* modeling the handling of a request for compensation within an airline. The set of traces that can be generated by this model is identical to the traces generated by the WF-net in Fig. 3.2 (the two models are trace equivalent). The inner nodes of the process tree represent operators. The leaves represent activities. There is one root node. Figure 3.17 shows the four types of operators can be used in a process tree: $\rightarrow$ (sequential composition), $\times$ (exclusive choice), $\wedge$ (parallel composition), and $\circlearrowleft$ (redo loop).

A sequence operator executes its children in sequential order. Activity a is the first child of the root node in Fig. 3.17. Since this node is a sequence node, every

process instance starts with activity a followed by the subtree starting with the redo loop ($\circlearrowleft$). After this subtree in the middle, the rightmost subtree is executed. The latter subtree models a choice ($\times$) between g and h.

The process tree in Fig. 3.17 can also be represented textually:

$$\rightarrow\big(a, \circlearrowleft\big(\rightarrow\big(\wedge\big(\times(b,c),d\big),e\big),f\big),\times(g,h)\big)$$

The rightmost subtree modeling the choice between activities g and h is represented as $\times(g,h)$. The redo loop $\circlearrowleft(\rightarrow(\wedge(\times(b,c),d),e),f)$ starts with its leftmost child and may loop back through any of its other children. In the process tree of Fig. 3.17, it is possible to loop back via "redo" activity f. The leftmost child ("do part") is $\rightarrow(\wedge(\times(b,c),d),e)$, i.e., a sequence that ends with activity e which is preceded by the subtree $\wedge(\times(b,c),d)$ where activity d is executed in parallel with a choice between b and c. The subtree $\wedge(\times(b,c),d)$ has four potential behaviors: $\langle b,d\rangle$, $\langle c,d\rangle$, $\langle d,b\rangle$, and $\langle d,c\rangle$.

The same activity may appear multiple times in the same process tree. For example, process tree $\rightarrow(a,a,a)$ models a sequence of three a activities. From a behavioral point of view, $\rightarrow(a,a,a)$ and $\wedge(a,a,a)$ are indistinguishable. Both have one possible trace, $\langle a,a,a\rangle$.

A silent activity is denoted by τ and cannot be observed. Process tree $\times(a,\tau)$ can be used to model an activity a that can be skipped. Process tree $\circlearrowleft(a,\tau)$ can be used to model the process that executes a at least once. The "redo" part is silent, so the process can loop back without executing any activity. Process tree $\circlearrowleft(\tau,a)$ models a process that executes a any number of times. The "do" part is now silent and activity a is in the "redo" part. This way it is also possible to not execute a at all. The smallest process tree is a tree consisting of just one activity. In this case the root node is also a leaf node and there are no operator nodes.

Definition 3.13 (Process tree) Let $A \subseteq \mathscr{A}$ be a finite set of activities with $\tau \notin A$. $\bigoplus = \{\rightarrow, \times, \wedge, \circlearrowleft\}$ is the set of *process tree operators*.

- If $a \in A \cup \{\tau\}$, then $Q = a$ is a process tree,
- If $n \geq 1$, $Q_1, Q_2, \ldots, Q_n$ are process trees, and $\oplus \in \{\rightarrow, \times, \wedge\}$, then $Q = \oplus(Q_1, Q_2, \ldots Q_n)$ is a process tree, and
- If $n \geq 2$ and $Q_1, Q_2, \ldots, Q_n$ are process trees, then $Q = \circlearrowleft(Q_1, Q_2, \ldots Q_n)$ is a process tree.

$\mathscr{Q}_A$ is the set of *all process trees* over A.

The redo loop operator $\circlearrowleft$ has at least two children. The first child is the "do" part and the other children are "redo" parts. Process tree $\circlearrowleft(a,b,c)$ allows for traces $\{\langle a\rangle, \langle a,b,a\rangle, \langle a,c,a\rangle, \langle a,b,a,b,a\rangle, \langle a,c,a,c,a\rangle, \langle a,c,a,b,a\rangle, \langle a,b,a,c,a\rangle, \ldots\}$. Activity a is executed at least once and the process always starts and ends with a. The "do" part alternates with the "redo" parts b or c. When looping back either b or c is executed.

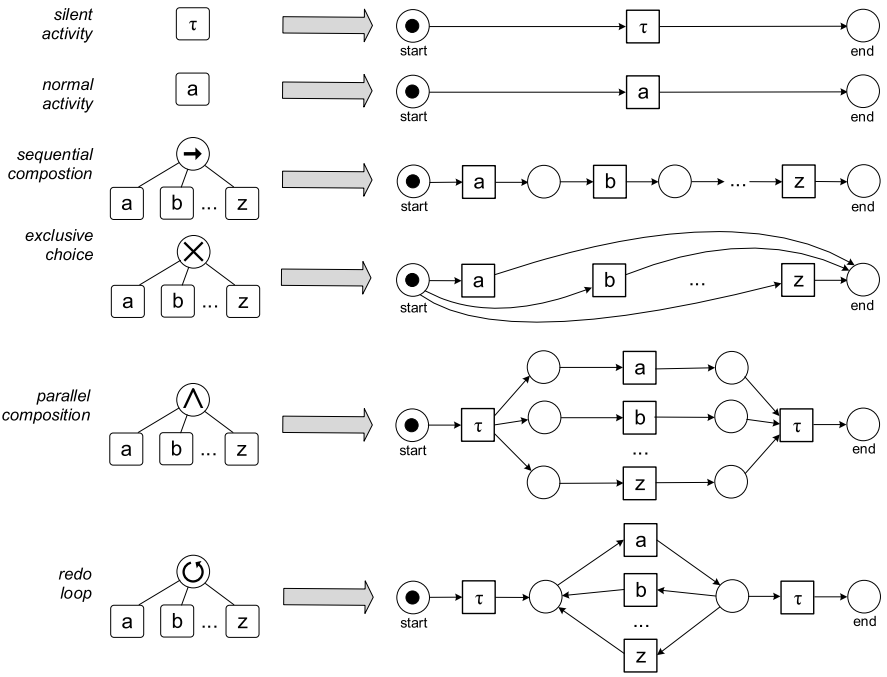


Fig. 3.18 Mapping process trees onto WF-nets

The redo loop operator $\circlearrowleft$ is often used in conjunction with silent activity τ. For example, $\circlearrowleft(\tau, a, b, c, \ldots, z)$ allows for any "word" involving activities $a, b, c, \ldots, z$. Example traces are $\langle\ \rangle$, $\langle a, b, b, a\rangle$, and $\langle w, o, r, d\rangle$.

Process trees can be converted to WF-nets as shown in Fig. 3.18. A silent activity is mapped onto a transition having a τ label. The mappings for $\rightarrow$ (sequential composition), $\times$ (exclusive choice), and $\wedge$ (parallel composition) are fairly straightforward. Silent transitions are used to model the start and end of the parallel composition. This is done to preserve the WF-net structure. The redo loop ($\circlearrowleft$) has one "do" part (activity a in Fig. 3.18) and one or more "redo" parts (activities b until z in Fig. 3.18). The direction of the arcs in the Petri net show the difference in semantics between the "do" and "redo" parts. Silent transitions are used to model the entry and exit of the redo loop. The mapping in Fig. 3.18 can be applied recursively and used to transform any process tree into a *sound* WF-net.

The mapping in Fig. 3.18 can easily be adapted for other representations such as BPMN, YAWL, EPCs, UML activity diagrams, statecharts, etc. The structured nature of process trees makes the conversion to other modeling notations straightforward. Conversions in the other direction (for example, from non-block-structured models to process trees) are more involved, but also less relevant since we only use process trees for process discovery. The mapping from process trees to WF-nets allows us to use existing conformance checking and performance analysis techniques.

The semantics of process trees can also be defined directly (without a mapping to WF-nets). To do this we first define two operators on sequences, concatenation ($\cdot$) and shuffle ($\diamond$).

Let $\sigma_1, \sigma_2 \in A^*$ be two sequences over A. $\sigma_1 \cdot \sigma_2 \in A^*$ *concatenates* two sequences, e.g., $\langle w, o \rangle \cdot \langle r, d \rangle = \langle w, o, r, d \rangle$. Concatenation can be generalized to sets of sequences. Let $S_1, S_2, \ldots, S_n \subseteq A^*$ be sets of sequences over A. $S_1 \cdot S_2 = \{\sigma_1 \cdot \sigma_2 \mid \sigma_1 \in S_1 \ \wedge \ \sigma_2 \in S_2\}$. For example, $\{\langle w, o \rangle, \langle\ \rangle\} \cdot \{\langle r, d \rangle, \langle k \rangle\} = \{\langle w, o, r, d \rangle, \langle w, o, k \rangle, \langle r, d \rangle, \langle k \rangle\}$. $\bigodot_{1 \leq i \leq n} S_i = S_1 \cdot S_2 \cdots S_n$ concatenates an ordered collection of sets of sequences.

$\sigma_1 \diamond \sigma_2$ generates the set of all interleaved sequences (shuffle). For example, $\langle w, o \rangle \diamond \langle r, d \rangle = \{\langle w, o, r, d \rangle, \langle w, r, o, d \rangle, \langle r, w, o, d \rangle, \langle w, r, d, o \rangle, \langle r, w, d, o \rangle, \langle r, d, w, o \rangle\}$. Note that the ordering in the original sequences is preserved, e.g., d cannot appear before r. Another example is $\langle w, o, r \rangle \diamond \langle d \rangle = \{\langle w, o, r, d \rangle, \langle w, o, d, r \rangle, \langle w, d, o, r \rangle, \langle d, w, o, r \rangle\}$. The shuffle operator can also be generalized to sets of sequences. $S_1 \diamond S_2 = \{\sigma \in \sigma_1 \diamond \sigma_2 \mid \sigma_1 \in S_1 \ \wedge \ \sigma_2 \in S_2\}$. The shuffle operator is commutative and associative, i.e., $S_1 \diamond S_2 = S_2 \diamond S_1$ and $(S_1 \diamond S_2) \diamond S_3 = S_1 \diamond (S_2 \diamond S_3)$. We write $\diamondsuit_{1 \leq i \leq n} S_i = S_1 \diamond S_2 \diamond \cdots \diamond S_n$ to interleave sets of sequences.

Definition 3.14 (Semantics) Let $Q \in \mathscr{Q}_A$ be a process tree over A. $\mathscr{L}(Q)$ is the *language* of Q, i.e., the set of traces that can be generated by it. $\mathscr{L}(Q)$ is defined recursively:

- $\mathscr{L}(Q) = \{\langle a \rangle\}$ if $Q = a \in A$,
- $\mathscr{L}(Q) = \{\langle\ \rangle\}$ if $Q = \tau$,
- $\mathscr{L}(Q) = \bigodot_{1 \leq i \leq n} \mathscr{L}(Q_i)$ if $Q = \rightarrow(Q_1, Q_2, \ldots, Q_n)$,
- $\mathscr{L}(Q) = \bigcup_{1 \leq i \leq n} \mathscr{L}(Q_i)$ if $Q = \times(Q_1, Q_2, \ldots, Q_n)$,
- $\mathscr{L}(Q) = \diamondsuit_{1 \leq i \leq n} \mathscr{L}(Q_i)$ if $Q = \wedge(Q_1, Q_2, \ldots, Q_n)$,
- $\mathscr{L}(Q) = \{\sigma_1 \cdot \sigma'_1 \cdot \sigma_2 \cdot \sigma'_2 \cdots \sigma_m \in A^* \mid m \geq 1 \ \wedge \ \forall_{1 \leq j \leq m} \ \sigma_j \in \mathscr{L}(Q_1) \ \wedge \ \forall_{1 \leq j < m} \ \sigma'_j \in \bigcup_{2 \leq i \leq n} \mathscr{L}(Q_i)\}$ if $Q = \circlearrowleft(Q_1, Q_2, \ldots, Q_n)$.

The following examples further illustrate the process tree operators and their semantics:

- $\mathscr{L}(\tau) = \{\langle\ \rangle\}$,
- $\mathscr{L}(a) = \{\langle a \rangle\}$,
- $\mathscr{L}(\rightarrow(a, b, c)) = \{\langle a, b, c \rangle\}$,
- $\mathscr{L}(\times(a, b, c)) = \{\langle a \rangle, \langle b \rangle, \langle c \rangle\}$,
- $\mathscr{L}(\wedge(a, b, c)) = \{\langle a, b, c \rangle, \langle a, c, b \rangle, \langle b, a, c \rangle, \langle b, c, a \rangle, \langle c, a, b \rangle, \langle c, b, a \rangle\}$,
- $\mathscr{L}(\circlearrowleft(a, b, c)) = \{\langle a \rangle, \langle a, b, a \rangle, \langle a, c, a \rangle, \langle a, b, a, c, a \rangle, \langle a, c, a, b, a \rangle, \ldots\}$,
- $\mathscr{L}(\rightarrow(a, \times(b, c), \wedge(a, a))) = \{\langle a, b, a, a \rangle, \langle a, c, a, a \rangle\}$,
- $\mathscr{L}(\times(\tau, a, \tau, \rightarrow(\tau, b), \wedge(c, \tau))) = \{\langle\ \rangle, \langle a \rangle, \langle b \rangle, \langle c \rangle\}$, and
- $\mathscr{L}(\circlearrowleft(a, \tau, c)) = \{\langle a \rangle, \langle a, a \rangle, \langle a, a, a \rangle, \langle a, c, a \rangle, \langle a, a, c, a \rangle, \langle a, c, a, c, a \rangle, \ldots\}$.

Process trees are *sound by construction*. Process discovery algorithms may exploit this when searching for a process model describing the event data. There are some similarities with other notations. *Process calculi* such as CSP and CCS use similar operators to model processes. Process trees can be viewed as a carefully

chosen subset. *Regular expressions* can model regular languages, e.g., $a^*(b|c)d^*$ denotes the set of traces starting with zero or more a's, followed by b or c, followed by zero or more d's. Process trees are in-between process calculi and regular expressions, and are tailored towards process discovery. Process calculi can handle concurrency, but are difficult to discover from event data (unless a similar subset is chosen). Regular expressions do not provide operators for concurrency and redo loops. However, in terms of expressiveness, process trees are comparable to regular expressions. Process trees are also related to *soundness preserving reduction rules* for Petri nets [168]. Reductions rules are normally used to reduce the size of a Petri net while preserving essential properties (e.g., soundness, liveness, boundedness, etc.). Starting from a WF-net with one transition, they can also be applied in reverse direction to produce larger sound WF-nets.

Section 7.5 introduces inductive process discovery techniques. Then the rationale for the choice of operators will become clearer. For example, $\circlearrowleft(\tau, a, b, c, \ldots, z)$ will be used as a last resort when all other operators are not applicable.

3.3 Model-Based Process Analysis

In Sect. 2.1, we discussed the different reasons for making models. Figure 2.4 illustrated the use of these models in the BPM life-cycle. Subsequent analysis showed that existing approaches using process models ignore event data. In later chapters we will show how to exploit event data when analyzing processes and their models. However, before doing so, we briefly summarize mainstream approaches for model-based analysis: *verification* and *performance analysis*. Verification is concerned with the correctness of a system or process. Performance analysis focuses on flow times, waiting times, utilization, and service levels.

3.3.1 Verification

In Sect. 3.2.3, we introduced the notion of soundness for WF-nets. This is a correctness criterion that can be checked using verification techniques. Consider, for example, the WF-net shown in Fig. 3.19. The model has been extended to model that *check ticket* should wait for the completion of *examine casually* but not for *examine thoroughly*. Therefore, place *c6* was added to model this dependency. However, a modeling error was made. One of the requirements listed in Definition 3.7, i.e., the "option to complete" requirement, is not satisfied. The marking $[c2, c3]$ is reached by executing the firing sequence $\langle a, b \rangle$ and from this marking the desired end marking $[end]$ is no longer reachable. Note that $[c2, c3]$ is a dead marking, e.g., d is not enabled because $c6$ is empty.

Definition 3.12 defines a soundness notion for C-nets. The notion of soundness can easily be adapted for other languages such as YAWL, EPCs, and BPMN. When

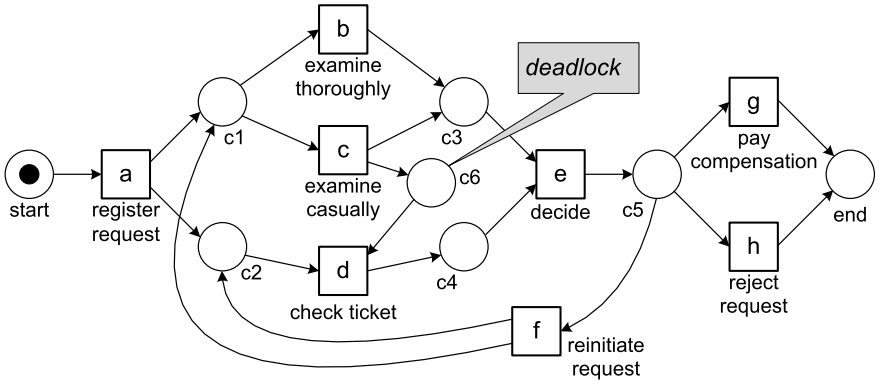


Fig. 3.19 A WF-net that is not sound

defining transition systems we already mentioned $S^{end} \subseteq S$ as the set of acceptable final states. Hence, we can define soundness as follows: a transition system is sound if and only if from any reachable state it is possible to reach a state in S^{end}. When introducing Petri nets we also defined generic properties such as liveness and boundedness. Some of these properties can be analyzed without constructing the state space. For example, for free-choice Petri nets, i.e., processes where choice and synchronization can be separated, liveness and boundedness can be checked by analyzing the rank of the corresponding incidence matrix [45]. Hence, soundness can be checked in polynomial time for free-choice WF-nets. Invariants can often be used to show boundedness or the unreachability of a particular marking. However, most of the more interesting verification questions require the exploration of (a part of) the state space.

Soundness is a generic property. Sometimes a more specific property needs to be investigated, e.g., "the ticket was checked for all rejected requests". Such properties can be expressed in *temporal logic* [30, 93]. *Linear Temporal Logic* (LTL) is an example of a temporal logic that, in addition to classical logical operators, uses temporal operators such as: always ($\Box$), eventually ($\Diamond$), until ($\sqcup$), weak until (W), and next time ($\bigcirc$). The expression $\Diamond h \Rightarrow \Diamond d$ means that for all cases in which h (*reject request*) is executed also d (*check ticket*) is executed. Another example is $\Box(f \Rightarrow \Diamond e)$ that states that any occurrence of f will be followed by e. *Model checking* techniques can be used to check such properties [30].

Another verification task is the comparison of two models. For example, the implementation of a process needs is compared to the high-level specification of the process. As indicated before, there exist different equivalence notions (trace equivalence, branching bisimilarity, etc.) [176]. Moreover, there are also various simulation notions demanding that one model can "follow all moves" of the other but not vice versa (see also Sect. 6.3).

There are various tools to verify process models. A classical example is Woflan that is tailored towards checking soundness [179]. Also workflow systems such as YAWL provide verification capabilities. Consider, for example, the screenshot

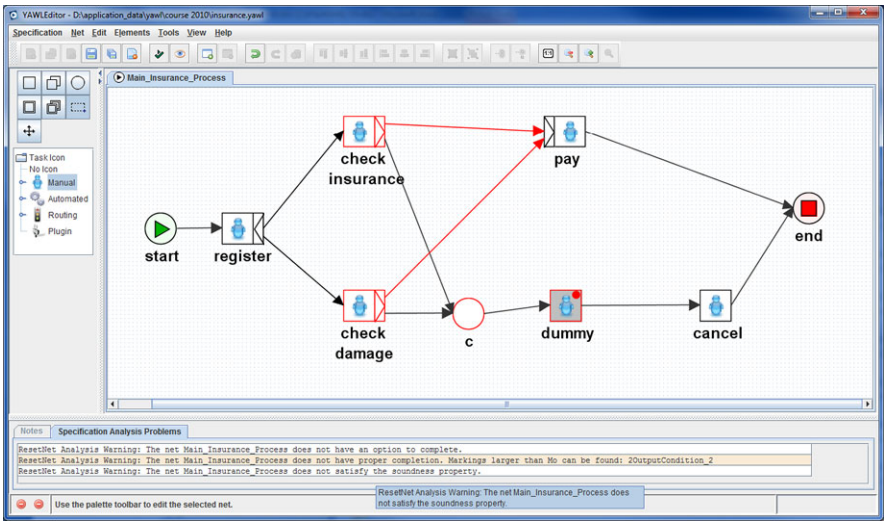


Fig. 3.20 An incorrect YAWL model: the cancelation region of *dummy* comprises of *check insurance*, *check damage*, condition *c* and the two implicit input conditions of *pay*. Hence, after cancelation, a token may be left on one of the output arcs of *register*

shown in Fig. 3.20. The figure shows the editor of YAWL while analyzing the model depicted. The process starts with task *register*. After this task, two checks can be done in parallel: *check insurance* and *check damage*. These tasks are XOR-splits; depending on the result of the check, one of the output arcs is selected. If both checks are OK, task *pay* is executed. If one of the checks indicates a problem, then the *dummy* task is executed. This task has a cancelation region consisting of *check insurance*, *check damage*, condition *c* and the two implicit input conditions of *pay*. The goal of this region is to remove all tokens, cancel the claim, and then end. However, the verifier of YAWL reports a problem. The YAWL model is not correct, because there may a be token pending in one of the implicit output conditions of *register*, i.e., there may be still a token on the arc connecting *register* and *check insurance* or on the arc connecting *register* and *check damage*. As a result the model may deadlock and "garbage" may be left behind. When these two implicit conditions are included in the cancelation region of the *dummy* task, then the verifier of YAWL will not find any problems and the model is indeed free of deadlocks and other anomalies.

3.3.2 Performance Analysis

The performance of a process or organization can be defined in different ways. Typically, three dimensions of performance are identified: *time*, *cost* and *quality*. For each of these performance dimensions different *Key Performance Indicators* (KPIs)

can be defined. When looking at the *time dimension* the following performance indicators can be identified:

- The *lead time* (also referred to as flow time) is the total time from the creation of the case to the completion of the case. In terms of a WF-net, this is the time it takes to go from source place i to sink place o. One can measure the average lead time over all cases. However, the degree of variance may also be important, i.e., it makes a difference whether all cases take more or less two weeks or if some take just a few hours whereas others take more than one month. The *service level* is the percentage of cases having a lead time lower than some threshold value, e.g., the percentage of cases handled within two weeks.
- The *service time* is the time actually worked on a case. One can measure the service time per activity, e.g., the average time needed to make a decision is 35 minutes, or for the entire case. Note that in case of concurrency the overall service time (i.e., summing up the times spent on the various activities) may be longer than the lead time. However, typically the service time is just a fraction of the lead time (minutes versus weeks).
- The *waiting time* is the time a case is waiting for a resource to become available. This time can be measured per activity or for the case as a whole. An example is the waiting time for a customer who wants to talk to a sales representative. Another example is the time a patient needs to wait before getting a knee operation. Again one may be interested in the average or variance of waiting times. It is also possible to focus on a service level, e.g., the percentage of patients that has a knee operation within three weeks after the initial diagnosis.
- The *synchronization time* is the time an activity is not yet fully enabled and waiting for an external trigger or another parallel branch. Unlike waiting time, the activity is not fully enabled yet, i.e., the case is waiting for synchronization rather than a resource. Consider, for example, a case at marking $[c2, c3]$ in the WF-net shown in Fig. 3.2. Activity e is waiting for *check ticket* to complete. The difference between the arrival time of the token in condition $c4$ and the arrival time of the token in condition $c3$ is the synchronization time.

Performance indicators can also be defined for the *cost dimension*. Different costing models can be used, e.g., Activity Based Costing (ABC), Time-Driven ABC, and Resource Consumption Accounting (RCA) [31]. The costs of executing an activity may be fixed or depend on the type of resource used, its utilization, or the duration of the activity. Resource costs may depend on the utilization of resources. A key performance indicator in most processes is the *average utilization* of resources over a given period, e.g., an operating room in a hospital has been used 85% of the time over the last two months. A detailed discussion of the various costing models is outside of the scope of this book.

The *quality dimension* typically focuses on the "product" or "service" delivered to the customer. Like costs, this can be measured in different ways. One example is customer satisfaction measured through questionnaires. Another example is the average number of complaints per case or the number of product defects.

Whereas verification focuses on the (logical) correctness of the modeled process, performance analysis aims at improving processes with respect to time, cost,

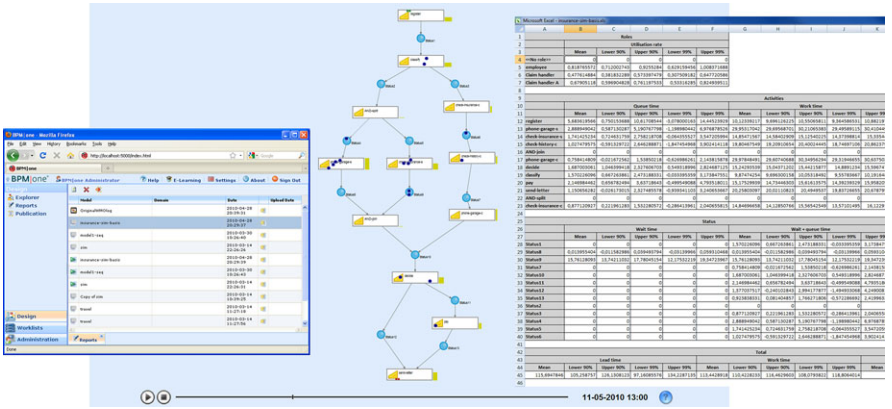


Fig. 3.21 Simulation using BPM|one of Pallas Athena: the modeled process can be animated and all kinds of KPIs of the simulated process are measured and stored in a spreadsheet

or quality. Within the context of operations management many analysis techniques have been developed. Some of these techniques "optimize" the model given a particular performance indicator. For example, integer programming or Markov decision problems can be used to find optimal policies. For the types of process models described in this chapter "what if" analyses using simulation, queueing models, or Markov models are most appropriate. Analytical models typically require many assumptions and can only be used to answer particular questions. Therefore, one needs to resort to *simulation*. Most BPM tools provide simulation capabilities. Figure 3.21 shows a screenshot of BPM|one while simulating a process for handling insurance claims. BPM|one can animate the simulation run and calculate all kinds of KPIs related to time and cost (e.g., lead time, service time, waiting time, utilization, and activity costs).

Although many organizations have tried to use simulation to analyze their business processes at some stage, *few are using simulation in a structured and effective manner*. This may be caused by a lack of training and limitations of existing tools. However, there are also several additional and more fundamental problems. First of all, simulation models tend to *oversimplify* things. In particular the behavior of resources is often modeled in a rather naïve manner. People do not work at constant speeds and need to distribute their attention over multiple processes. This can have dramatic effects on the performance of a process and, therefore, such aspects should not be "abstracted away" [139, 163]. Second, various *artifacts available are not used as input for simulation*. Modern organizations store events in logs and some may have accurate process models stored in their BPM/WFM systems. Also note that in many organizations, the state of the information system accurately reflects the state of the business processes supported by these systems. As discussed in Chap. 1, processes and information systems have become tightly coupled. Nevertheless, such information (i.e., event logs and status data) is rarely used for simulation or a lot of manual work is needed to feed this information into the model. Fortunately, as will

be shown later in this book, process mining can assist in extracting such information and use this to realize performance improvements (see Sect. 9.6). Third, the focus of simulation is mainly on "design" whereas managers would also like to use simulation for *"operational decision making"*, i.e., solving the concrete problem at hand rather than some abstract future problem. Fortunately, *short-term simulation* [139] can provide answers for questions related to "here and now". The key idea is to start all simulation runs from the current state and focus on the analysis of the transient behavior. This way a "fast forward button" into the future is provided.

3.3.3 Limitations of Model-Based Analysis

Verification and performance analysis heavily rely on the availability of high quality models. When the models and reality have little in common, model-based analysis does not make much sense. For example, the process model can be internally consistent and satisfy all kinds of desirable properties. However, if the model describes an idealized version of reality, this is quite useless as in reality all kinds of deviations may take place. Similar comments hold for simulation models. It may be that the model predicts a significant improvement whereas in reality this is not the case because the model is based on flawed assumptions. All of these problems stem from *a lack of alignment between hand-made models and reality.* Process mining aims to address these problems by establishing a direct connection between the models and actual low-level event data about the process. Moreover, the *discovery techniques discussed in this book allow for viewing the same reality from different angles and at different levels of abstraction*.