Synthesis of Petri Nets for Business Process Design

Robin Bergenthum, Jörg Desel, Sebastian Mauser *

firstname.lastname@ku-eichstaett.de

Abstract: We propose the following approach to model a business process applying Petri net synthesis: In a first phase runs supported by the process are designed. The second phase is automatically creating a Petri net model of the business process using synthesis algorithms. We present a concrete procedure including several design steps for the first phase. For the second phase we develop an appropriate adaption of existing synthesis algorithms.

1 Introduction

Business process modelling has attracted increasing attention in recent years [vdAvH02, vdADtH05, Wes07]. One of the main issues of modelling a business process is analysis. Analysis requires a formalization of the business process model. Obviously, the value of the analysis outcome depends on the correctness of the model with respect to the actual (or intended) business process. To avoid confusion with the formal interpretation of the term correctness (a model is correct if it satisfies a property formulated by a specification), we call a model *valid* if it faithfully represents the business process. Most of the theory concerned with business process management and according tools assume validity of models and concentrate on analysis and verification issues. However, experience shows that in many cases negative results of analysis or verification are caused by invalid models rather than by incorrect business processes. Even worse, positive results do not mean much if validity of models cannot be assumed. Therefore, the first phase of modelling is of particular importance. Any error in an early stage of modelling will cause very costly redevelopment efforts in later phases.

The paper [Des08] systematically tackled the problems occurring in the beginning of process design. Its ideas are based on suggestions for a systematic analysis of requirements developed in [MS93, May98, MK02, MKE07, FMK03]. It is argued that knowledge about a process is typically distributed in different people's minds. Deriving a valid process model in such an informal environment is a difficult and error-prone task. The classical modelling approach starts with identifying tasks and resources of the process. Then the knowledge of all involved people is collected. This step is often supported by semi-formal modelling languages. The resulting process descriptions are used to design the control flow of the process model. The validity of the model is checked by examining

*Supported by the project "SYNOPS" of the German Research Council

its behaviour in comparison to the behaviour of the business process, either implicitly or explicitly [DJLN03, BDJL06].

The modelling approach proposed in [Des08] is different. The simplest modelling concept is on the instance level: a single *run* of a process. It is assumed that the relevant persons can more easily describe single runs than considering the process as a whole. In [SMG06, Gli95, Gli00, Des99] key advantages of using runs in requirements engineering are surveyed showing their intuitive handling. In particular, runs need not be designed by some modelling expert, but they may be designed independently by domain experts.

Therefore the novel design approach (shown in detail by the enumeration below) starts by collecting runs, also called scenarios or use cases, of the business process. The development of a specification of the business process in terms of runs includes several problems such as identifying relevant single activities. Having created such specification, we propose to use synthesis algorithms to generate automatically a process model. The problem of creating a process model from a respective behavioural specification is tackled by Petri net synthesis methods [BD96, LBDM07, BLM06] and process mining techniques [vdA07, BDLM07]. Petri nets are a standard formalism for the representation of processes [vdAvH02, vdADtH05].

In the described approach the complex task of integrating the different views of a process (in the standard modelling approach) is performed automatically by synthesis methods. The only task that has to be done "by hand" in this new modelling approach is to design appropriate single runs of the process. Assuming that the specification given by the collected runs has a high quality, a high degree of validity of the process model is ensured, since the automatically synthesized model conforms to the specified runs.

This design approach is embedded in [Des08] into a comprehensive business process modelling procedure allowing the refinement of single activities to subprocesses. While all other steps of this modelling procedure are explained in detail, the design steps of specifying runs of the process and of synthesizing a Petri net from the specified runs are only described on an abstract level. In this paper we fill this gap. We propose a flexible method to design runs of a process and show in detail how to synthesize a Petri net model of the process from the specification given by the runs.

The following enumeration shows the whole modelling procedure proposed in [Des08]. The considerations in this paper explain how to perform steps 2-4 and 7-9.

1. First, identify start conditions, start actions and end actions of the process to be defined.
2. Let relevant people define runs of the process on an abstract level.
3. Agree on the abstract actions that occur in these runs.
4. Synthesize a process from the runs using synthesis methods.
5. Validate this process by generating runs.
6. Identify actions that have to be refined.

7. Identify experts that can provide information (namely runs) for these actions.
8. Agree on actions that occur in these runs.
9. Synthesize a process from the runs using synthesis methods.
10. Validate this process by generating runs.
11. Iterate the procedure if there is a need for a higher refinement hierarchy.
12. Construct the entire flat process by refining all transitions on the abstract level (possibly to be repeated for the next levels).
13. Analyze this process w.r.t. appropriate correctness criteria, such as liveness, using known techniques for Petri nets.

The remainder of the paper is organized as follows: In Section 2 we describe a procedure of developing a specification in terms of runs. In Section 3 a synthesis approach to automatically generate a process model from the developed behavioural specification is presented. Section 4 illustrates the proposed business process design approach by a simple case study supported by our modelling tool VipTool [DJLN03, BDJL06].

The approach described in the following sections requires advanced techniques from the research field of partially ordered runs of Petri nets. We present the formalities in separate paragraphs distinguished by italic font. The technical paragraphs are announced by a bold faced "Technical part". These parts only regard technical details. Readers not familiar with the field of partially ordered runs of Petri nets may skip these parts for better readability.

2 Specifying Runs

The main task in the new design approach is specifying the behaviour of a process in terms of single runs. We claim that modelling a single run is an easy and intuitive task, also for domain experts unexperienced in modelling. There are many possibilities to describe runs (also textual descriptions are adequate). Using runs in requirements engineering has received significant attention in the last years [SMG06, Gli95, Gli00]. In particular, descriptions of runs occur in almost all modelling languages, e.g. activity diagrams, sequence diagrams and use case diagrams in the case of the UML language. Runs do not necessarily have to be designed from scratch. In many cases it is possible to exploit already existing descriptions of runs supported by the business process. In an enterprise, typical sources of scenario descriptions are log files recorded by information systems (process mining focuses on this source of information), process instructions for employees or textual and formal process descriptions from some requirements analysis.

In the first step of the design approach, as many as possible single runs of the process are identified to get a preferably complete description of the behaviour of the business process. In this paper, we consider the formal model of *labelled partial orders (LPOs)* to specify

single runs. LPOs are a very general formalism and most languages used in practice can be mapped to LPOs.

Technical part (labelled partial order) *A labeled partial order (LPO) is a triple* $\mathrm{lpo} = (V, <, l)$*, where* V *is a set of events,* $<$ *is an irreflexive and transitive binary relation on* V*, and* $l : V \to T$ *is a labeling function with set of labels* T*.*

The behaviour specified by an LPO includes so called prefixes and sequentializations. An LPO $(V', <', l')$ *is called a prefix of another LPO* $(V, <, l)$ *if* $V' \subseteq V$*,* $(v' \in V' \wedge v < v') \implies (v \in V')$*,* $<'=< \cap V' \times V'$ *and* $l' = l|_{V'}$*. An LPO* $(V, <', l)$ *is called a sequentialization of another LPO* $(V, <, l)$ *if* $< \subseteq <'$*. We consider LPOs only up to isomorphism (i.e. isomorphic LPOs are not distinguished), because isomorphic LPOs model the same behaviour. Two LPOs* $(V, <, l)$ *and* $(V', <', l')$ *are called isomorphic, if there is a bijective mapping* $\psi : V \to V'$ *such that* $l(v) = l'(\psi(v))$ *for* $v \in V$*, and* $v < w \iff \psi(v) <' \psi(w)$ *for* $v, w \in V$*.*

An LPO models a single run by specifying "earlier than"-dependencies (a causal order) between events, where events represent occurrences of respective activities. LPOs allow to account for arbitrary concurrency relations between activities. They offer the following advantages in business process modelling [DJLN03]:

- *A natural and intuitive representation of the behaviour of business processes:* Since concurrency plays an important role in business process models, it is appropriate to enable modelling of concurrency also in single runs of a business process. In particular, instead of considering sequential runs and trying to detect possible concurrency relations from a set of sequential runs, it is much easier and more intuitive to work with partially ordered runs.
- *An efficient representation of the behaviour of business processes:* A single LPO represents a set of sequential runs, which can be quite large (exponential in the number of transition occurrences) in the presence of concurrency.
- *A high degree of expressiveness:* First, considering sequential runs, concurrency cannot be distinguished from non-deterministic resource sharing. Second, LPOs explicitly model causal dependencies between transition occurrences, which enables the explicit modeling of the flow of objects and information in business processes (this is not even implicitly possible with sequential runs).
- *Efficient analysis algorithms for business process models:* In many cases analysis techniques applied to LPOs are more efficient than those considering sequential runs [BDJL06, BM07].

The second step of the approach is the translation of the various scenario descriptions to LPOs. Since LPOs allow modelling of true concurrency, such translation will always be possible. We do not discuss problems related with this step in this paper.

As a result of the second step we get a collection of LPOs representing runs of the process. An example of such a set of LPOs is shown in Figure 1 for the workflow triggered by a damage report in an insurance company (claim processing). The figure depicts a run

showing the registration process, several runs describing possible evaluation procedures of the claim, a run modelling the payment of the insurance company as well as the three singleton runs for building reserves, gathering information for the payment by asking queries and the completion of the workflow. It is assumed that the runs modelling the different aspects of the process have been described by various involved people. Since modelled runs typically only describe a certain part of the behaviour of the process, the problem is that the runs may have different relationships to each other.

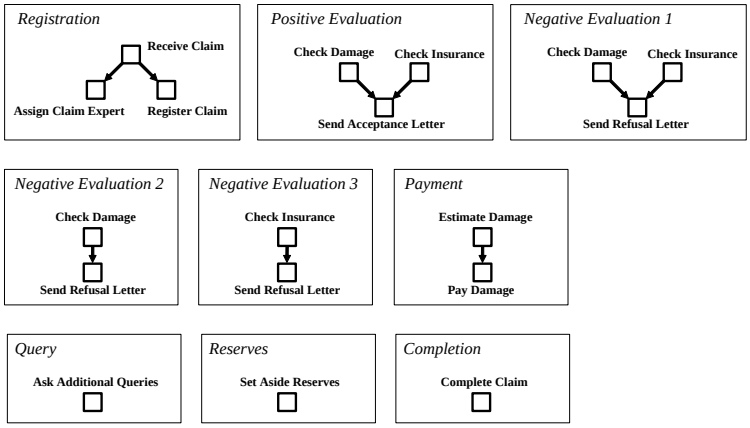

Figure 1: Example for a collection of runs in terms of LPOs.

The third step of our approach is to integrate the runs into a single model respecting the relations between the runs. Runs can be related in four ways (Figure 2):

- One specified run of the process can only occur after another run (sequence).
- Either one run can occur or another run (alternative).
- A specified run can be followed n times by itself (iteration).
- Two runs of the process can occur concurrently (concurrency).

Similar to the approach for scenario integration based on statecharts in [Gli95], higher level structures of runs are built by concatenating and nesting blocks according to the relationships sequence (;), alternative ($+$), iteration ($*$), and concurrency($||$). For the runs of Figure 1, we assume that they are related as depicted in Figure 2 to faithfully model the underlying workflow. That means the workflow starts with the ”Registration” of the claim. Then one of the evaluation runs is performed concurrently to the subprocess of setting aside reserves for the claim. The four possibilities of evaluation are alternatives. The run modelling the positive case is a sequential composition of the three single runs ”Positive Evaluation”, ”Queries” and ”Payment”. Asking additional queries can be iterated arbitrarily often until a sufficient degree of information is reached. Finally, after the execution of all other runs, the process is finished by the run ”Completion”. Figure 2 also

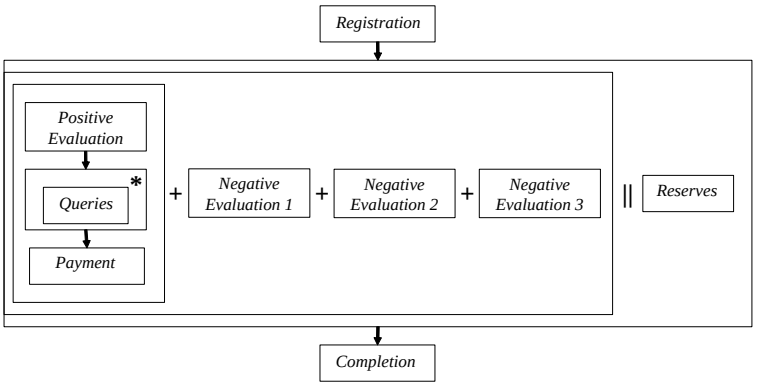

Figure 2: A composed run over the set of runs depicted in Figure 1.

illustrates a possible graphical representation of the four composition templates for runs (;-composition is depicted by arcs, $+$, $*$ and $||$ by respective symbols). While the composition operators are actually binary, the readability of the graphical representation can often be improved by composing more than two blocks in figures (e.g. the $+$-composition of the four alternative evaluation possibilities). We call such behavioural specification generated by the composition of single runs a *composed run*.

Since single runs are given by LPOs, the semantics of a composed run $\mathbf{R}$ is defined as a set of LPOs $\mathfrak{Lpo}(\mathbf{R})$ modelling alternative behaviour, called the *set of runs defined by a composed run*. The set of runs defined by the composed run of Figure 2 is illustrated in the screenshot of Figure 5 in Section 4. The iteration of the activity "Ask Additional Queries" generates an infinite set of runs. Figure 5 only shows the cases that this activity is iterated zero, one and two times (scenario 4-6). Runs with more iterations of the activity are analogous.

Technical part (composed run) *Given a finite set of single runs $\mathcal{A}$, a composed run over $\mathcal{A}$ is inductively defined as follows: The single runs $\mathrm{lpo} \in \mathcal{A}$, the empty LPO $\lambda = (\emptyset, \emptyset, \emptyset)$ and the empty set $\emptyset$ are composed runs. Let $\mathbf{R}_1$ and $\mathbf{R}_2$ be composed runs. Then $\mathbf{R} = \mathbf{R}_1; \mathbf{R}_2$ (sequential composition), $\mathbf{R} = \mathbf{R}_1 + \mathbf{R}_2$ (alternative composition), $\mathbf{R} = (\mathbf{R}_1)^*$ (iteration) and $\mathbf{R} = \mathbf{R}_1 \parallel \mathbf{R}_2$ (concurrent composition) are composed runs.*

We define for two LPOs $\mathrm{lpo}_1 = (V_1, <_1, l_1), \mathrm{lpo}_2 = (V_2, <_2, l_2) \in \mathcal{A}$ the sequential composition by $\mathrm{lpo}_1; \mathrm{lpo}_2 = (V_1 \cup V_2, <_1 \cup <_2 \cup (V_1 \times V_2), l_1 \cup l_2)$, the parallel composition by $\mathrm{lpo}_1 \parallel \mathrm{lpo}_2 = (V_1 \cup V_2, <_1 \cup <_2, l_1 \cup l_2)$, and denote $\mathrm{lpo}_1^0 = \lambda$ and $\mathrm{lpo}_1^n = \mathrm{lpo}_1^{n-1}\mathrm{lpo}_1$ for $n \in \mathbb{N}^+$ ($\mathbb{N}^+$ denotes the positive integers).[1]

The set of runs $\mathfrak{Lpo}(\mathbf{R})$ of a composed run $\mathbf{R}$ over $\mathcal{A}$ is a possibly infinite set of LPOs. Given a composed run $\mathbf{R}$, we first inductively define a set of LPOs $K(\mathbf{R})$ represented by $\mathbf{R}$. The set $\mathfrak{Lpo}(\mathbf{R})$ is the prefix and sequentialization closure of $K(\mathbf{R})$. We set $K(\lambda) = \{\lambda\}$, $K(\emptyset) = \emptyset$ and $K(\mathrm{lpo}) = \{\mathrm{lpo}\}$ for $\mathrm{lpo} \in \mathcal{A}$. We further define inductively for LPO-terms $\mathbf{R}_1$ and $\mathbf{R}_2$:

[1] We assume that $\mathrm{lpo}_1, \mathrm{lpo}_2$ have disjoint sets of nodes.

- $K(\mathbf{R}_1 + \mathbf{R}_2) = K(\mathbf{R}_1) \cup K(\mathbf{R}_2)$.
- $K(\mathbf{R}_1; \mathbf{R}_2) = \{\mathrm{lpo}_1 \mathrm{lpo}_2 \mid \mathrm{lpo}_1 \in K(\mathbf{R}_1),\, \mathrm{lpo}_2 \in K(\mathbf{R}_2)\}$.
- $K((\mathbf{R}_1)^*) = \{\mathrm{lpo}_1 \ldots \mathrm{lpo}_n \mid \mathrm{lpo}_1, \ldots, \mathrm{lpo}_n \in K(\mathbf{R}_1),\, n \in \mathbb{N}^+\} \cup \{\lambda\}$.
- $K(\mathbf{R}_1 \parallel \mathbf{R}_2) = \{\mathrm{lpo}_1 \parallel \mathrm{lpo}_2 \mid \mathrm{lpo}_1 \in K(\mathbf{R}_1),\, \mathrm{lpo}_2 \in K(\mathbf{R}_2)\}$.

In Figure 5, the set $K(\mathbf{R})$ (not regarding all iterations of the activity "Ask Additional Queries") of the composed run $\mathbf{R}$ depicted in Figure 2 is shown. The set of runs $\mathfrak{Lpo}(\mathbf{R})$ of $\mathbf{R}$ is given by the set of prefixes of sequentializations of LPOs in $K(\mathbf{R})$.

Integrating single runs to one composed run has to be done cooperatively by the involved practitioners. They have to detect the detailed relationships between the runs by exchanging their knowledge about dependencies of runs to other parts of the behaviour of the process. At the end of the third step of our approach they have to agree on one composed run including all single runs. The composed run represents an integrated model of the behaviour of the considered business process.

A problem excluded so far is that some runs may overlap. That means, the knowledge about one run may be distributed on several experts, each knowing only an *extract* of the whole run. In the simplest case extracts can be treated as single (sub-) runs that can be composed as shown above. But this is not possible if the extracts contain common events. Then the extracts of the respective run have to be fused to one single run. The situation of extracts having common events occurs if several experts have different views to one process execution, i.e. they observe different subsets of all events of the respective run, whereas respective other parts of the run are hidden.

We propose the following concept to *fuse extracts* of one run. Given several extracts of one run, first the involved people have to determine which events of an extract of the run observed by one expert coincide with events of an extract of the run observed by another expert. This problem has to be solved by an appropriate communication between the experts. They have to agree on a *fusion equivalence relation* on the set of events of all extracts of the considered run, such that different observations of one event are equivalent. Obviously, only events of different extracts having the same label (referring to the same activity) can be equivalent. Also, the orderings given by different extracts must not contradict each other. The fusion of the extracts is then given by a new LPO, which has an event for each equivalence class. If two events are ordered in some extract, then their respective classes are ordered in this LPO. Thus, each dependency observed (respectively modelled) by some expert in some extract of the run is regarded in the fused run. No further dependencies are introduced. Conversely, we assume concurrency between events if no expert detected any dependency. This is motivated by the idea that extracts of one run observed by different experts tend to be concurrent.

A simple example is shown in Figure 3. Assume the run "Registration" of Figure 1 is not given directly, but rather the two extracts of the run shown in Figure 3 are specified by two experts (responsible for respective tasks). If they agree that the two events labelled by "Receive Claim" coincide, the described fusion approach generates the fused run "Registration" of Figure 1.

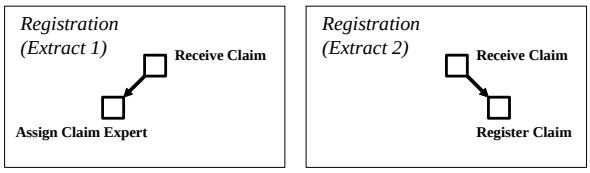

Figure 3: Two possible extracts of the run "Registration" shown in Figure 1.

Technical part (fusion) *Given LPOs* $\mathrm{lpo}_i = (V_i, <_i, l_i)$*,* $i \in \{1, \ldots, n\}$ *(all* V_i *can be assumed pairwise disjoint) modelling different extracts of one run and an equivalence relation* $\sim$ *on* $\bigcup_{i=1}^{n} V_i$*, fulfilling the fusion requirement* $v \sim v', v \neq v', v \in V_i, v' \in V_j \Longrightarrow i \neq j \wedge l_i(v) = l_j(v') \wedge \forall v'' \in V_i, v''' \in V_j, v'' \sim v''' : (v <_i v'' \Longrightarrow v''' \not<_j v')$*, the fused LPO of* $\mathrm{lpo}_i = (V_i, <_i, l_i)$*,* $i \in \{1, \ldots, n\}$ *w.r.t.* $\sim$ *is defined by* $\mathrm{lpo} = (V, <, l)$*, where* $V = \{[v]_\sim \mid v \in \bigcup_{i=1}^{n} V_i\}$*,* $[v]_\sim < [v']_\sim \Longleftrightarrow (\exists v'' \in [v]_\sim, v''' \in [v']_\sim, i \in \{1, \ldots, n\} : v'', v''' \in V_i, v'' <_i v''')$ *and* $l([v]_\sim) = l_i(v)$ *(for* $v \in V_i$*). The fused LPO is well defined by the fusion requirement.*

Finally, if the set of single runs given after step two of our approach still contains extracts of runs, such extracts either have to be considered as usual single runs or the extracts have to be fused to one single run as shown above. Having done this at the beginning of step three of the approach, the requirements to define a composed run as described above are fulfilled.

3 Synthesizing a Process Model

The next step in the design approach starts with a specification of the business process by a composed run. The aim is to automatically create a Petri net model from the composed run. Formally, the composed run is defined as a term over the alphabet of single runs employing the composition operators ;, $+$, $*$ and $||$. In a recent submission to the ACSD2008 conference (based on [BLM06]), we show how to synthesize a *place/transition net (p/t-net)* from such a term. A synthesis algorithm is presented which computes from a composed run a p/t-net having the specified behaviour and minimal additional behaviour. That means, the set of runs of the computed p/t-net is the *best upper approximation* (possible by a p/t-net) to the set of runs of the composed run.

The activities of the business process are modelled by the transitions of the synthesized Petri net. The places together with their connections to the transitions and their markings define dependencies between the activities. Figure 6 shows a marked p/t-net. As usual, places are drawn as circles including tokens representing the initial marking, transitions as rectangles and the flow relation as arcs annotated with values of the weight function (arcs with weight 0 are not drawn, the weight 1 is not shown).

A *run of a p/t-net* (N, m_0) is given by an LPO with event labels referring to transitions, such that the events can occur respecting the concurrency and dependency relations of the

LPO. Thus, a run describes executable behaviour of the net in the sense that the transition occurrences given by the events are possible in the net only using the dependencies specified by the run for the flow of tokens. The set of all runs of (N, m_0) is denoted by $\mathfrak{Lpo}(N, m_0)$.

The synthesized p/t-net (N, m_0) is a best upper approximation to the specified composed run $\mathbf{R}$ in the sense that $\mathfrak{Lpo}(\mathbf{R}) \subseteq \mathfrak{Lpo}(N, m_0)$ and $\forall (N', m_0') : (\mathfrak{Lpo}(\mathbf{R}) \subseteq \mathfrak{Lpo}(N', m_0')) \Longrightarrow (\mathfrak{Lpo}(N, m_0) \subseteq \mathfrak{Lpo}(N', m_0'))$.

Technical part (p/t-net) *A (marked) p/t-net is a quadruple (P, T, W, m_0), where P is a finite set of places, T is a finite set of transitions satisfying $P \cap T = \emptyset$, $W : (P \times T) \cup (T \times P) \to \mathbb{N}^+$ is a weight function defining the flow relation, and $m_0 : P \to \mathbb{N} \in \mathbb{N}^P$ ($\mathbb{N}$ denotes the non-negative integers) is an initial marking.*

The behaviour of a p/t-net is defined as follows: A multi-set of transitions $\tau : T \to \mathbb{N} \in \mathbb{N}^T$ is called a step *(of transitions). A step τ is enabled to occur (concurrently) in a marking $m : P \to \mathbb{N} \in \mathbb{N}^P$ of a p/t-net, if and only if $m(p) \geq \sum_{t \in \tau} \tau(t) W(p,t)$ for each place $p \in P$. In this case, its occurrence leads to the marking $m'(p) = m(p) + \sum_{t \in \tau} \tau(t)(W(t,p) - W(p,t))$, abbreviated by $m \xrightarrow{\tau} m'$. A finite sequence of steps $\sigma = \tau_1 \ldots \tau_n,\ n \in \mathbb{N}$, is called a step occurrence sequence enabled in a marking m and leading to m_n, denoted by $m \xrightarrow{\sigma} m_n$, if there exists a sequence of markings $m_1, \ldots, m_n$ such that $m \xrightarrow{\tau_1} m_1 \xrightarrow{\tau_2} \ldots \xrightarrow{\tau_n} m_n$.*

We are interested in the behaviour in terms of LPOs. Given an LPO $\mathrm{lpo} = (V, <, l)$, two events $v, v' \in V$ are called independent if $v \not< v'$ and $v' \not< v$, denoted by $v \,\mathrm{co}\, v'$. A co-set is a subset $C \subseteq V$ fulfilling: $\forall v, v' \in C : v \,\mathrm{co}\, v'$. A cut is a maximal co-set. For a co-set C and an event $v \in V \setminus C$ we write $v < (>) C$, if $v < (>) v'$ for an element $v' \in C$ and $v \,\mathrm{co}\, C$, if $v \,\mathrm{co}\, v'$ for all elements $v' \in C$. Given a marked p/t-net (N, m_0), an LPO $\mathrm{lpo} = (V, <, l)$ with $l : V \to T$ is called a run of (N, m_0) if $m_0(p) + \sum_{v \in V \wedge v < C} (W(l(v), p) - W(p, l(v))) \geq \sum_{v \in C} W(p, l(v))$ for every cut C of lpo and every $p \in P$. The set of runs of a p/t-net (N, m_0) is defined by $\mathfrak{Lpo}(N, m_0) = \{(V, <, l) \mid (V, <, l)$ is a run of $(N, m_0)\}$.

Synthesizing an upper approximation is useful, because the behaviour explicitly specified by $\mathbf{R}$ should definitely be included in the behaviour of the synthesized model. The best upper approximation property ensures that only necessary additional behaviour is added to the synthesized net. Thus, computing a best upper approximation may be seen as a natural completion of the specified behaviour $\mathbf{R}$ by a Petri net. Such an approach is necessary, since process specifications in practice are typically incomplete and not arbitrary specified behaviour can be reproduced by a Petri net.

In the underlying design procedure of [Des08], a special class of Petri nets is considered to model business processes: We consider connected p/t-nets with two distinguished sets of input and output transitions as *processes*. In Figure 4 such processes are shown, where the input respectively output transitions are depicted with two ingoing respectively outgoing arcs.

Actually, in [Des08] no arc weights are considered. Moreover, the nets are required to have a certain 1-boundedness property. This property ensures that the refinement steps of the design procedure can properly be performed. In a recent submission to ATPN2008, we show

how this property can be guaranteed by the considered synthesis algorithm. Nevertheless, in the present paper we consider the definition of processes based on general p/t-nets, because the synthesis algorithm is a lot more efficient in this case [LBDM07, BDLM07]. To still allow the same *refinement* procedure for transitions as shown in [Des08], it has to be ensured that never a second instance of some subprocess is started (by an initial transition) before a prior instance of the same subprocess is finished (by a final transition). This can be achieved by adding to a transition t, before it is refined, a self-loop place p_t with $W(p_t, t) = W(t, p_t) = m_0(p_t) = 1$. This place in particular prohibits auto-concurrency. Refining t by a subprocess, the place p_t has an outgoing arc with weight one to every input transition of a subprocess and an ingoing arc with weight one from every output transition of a subprocess (see Figure 4). Then input and output transitions of the subprocess occur strictly alternatingly, i.e. it is not possible that two input transitions occur without an intermediate output transition.

Formally, the refinement steps in our setting are defined as follows: We consider one main process. A subprocess refines a transition (to which a self-loop place was added) of another process (either of the main process or of another subprocess). It replaces the transition, the transition's input places are connected to the input transitions of the subprocess with arcs having the same weights, whereas the output transitions are connected to the output places of the transition by arcs having the same weights (see Figure 4). Hence, the external behaviour of a subprocess resembles the behaviour of a transition, with the difference that first the input tokens are consumed and later the output tokens are produced. The order of refinement does not matter.

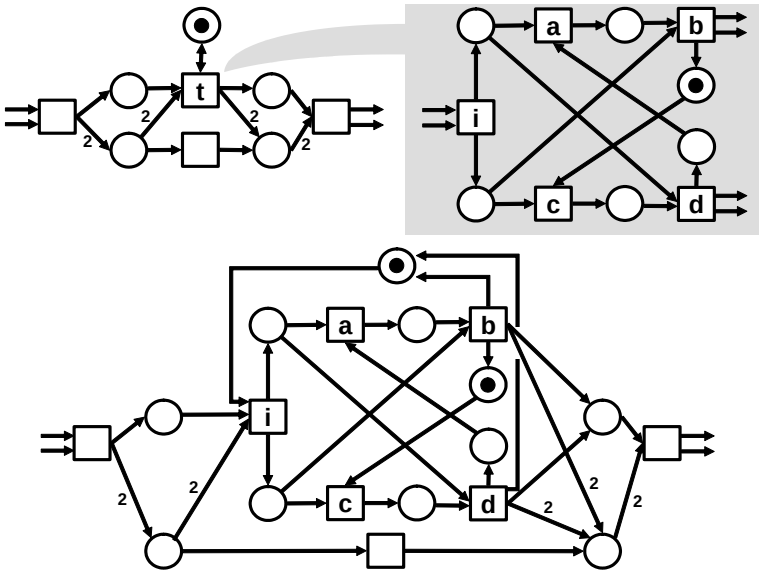

Figure 4: Top: Two abstract processes. Bottom: The refinement of the left process w.r.t. t and the right process.

Technical part (refinement) *A process is a connected p/t-net* (P, T, W, m_0) *with two distinguished sets of input and output transitions* $T_i, T_o \subseteq T$.

Let $N = (P, T, W, m_0)$ *be a process with a transition* t *and let* $N^t = (P^t, T^t, W^t, m_0^t)$ *be a process with input transitions* T_i^t *and output transitions* T_o^t. *Assume w.l.o.g. that the elements of* P *and of* P^t *as well as of* T *and of* T^t *are disjoint. The refinement of* N *w.r.t. transition* t *and process* N^t *is defined as* $(P \cup P^t \cup \{p_t\}, T \cup T^t \setminus \{t\}, W_{N,N^t}, m_0 \cup m_0^t)$, *where* W_{N,N^t} *is defined by* $W_{N,N^t}|_{P\times(T\setminus\{t\})\cup(T\setminus\{t\})\times P} = W|_{P\times(T\setminus\{t\})\cup(T\setminus\{t\})\times P}$, $W_{N,N^t}|_{P^t\times T^t\cup T^t\times P^t} = W^t$, $W_{N,N^t}(p, t_i) = W(p, t), W_{N,N^t}(t_o, p) = W(t, p)$ *for* $p \in P, t_i \in T_i^t, t_o \in T_o^t, W(p_t, t_i) = W(t_o, p_t) = 1$ *for* $t_i \in T_i^t, t_o \in T_o^t$, *and* $W_{N,N^t} = 0$ *else.*

It only remains to tune the considered synthesis algorithm to the definition of processes regarded in our approach, i.e. the sets of input and output transitions have to be regarded in the synthesis algorithm to create reasonable process models.

That means, from the set of activities of the process, firstly the sets of input transitions T_i and output transitions T_o have to be defined. To ensure that a process starts with an input transition, finishes with an output transition and in between no input and output transitions occur, we require for the specification given by the composed run $\mathbf{R}$ that

- every non-empty LPO $(V, <, l)$ in the set of runs $\mathfrak{Lpo}(\mathbf{R})$ of $\mathbf{R}$ contains exactly one event $v_0 \in V$ labelled by an input transition of the process ($l(v_0) \in T_i$). The event v_0 is called initial event.
- the initial event v_0 of every non-empty LPO $(V, <, l) \in \mathfrak{Lpo}(\mathbf{R})$ is a unique minimal event, i.e. it fulfills $v_0 < v$ for every $v \in V \setminus \{v_0\}$,
- every LPO $(V, <, l)$ in the set of runs $\mathfrak{Lpo}(\mathbf{R})$ of $\mathbf{R}$, which is not prefix of another LPO in $\mathfrak{Lpo}(\mathbf{R})$, called maximal LPO of $\mathfrak{Lpo}(\mathbf{R})$, contains exactly one event $v_{max} \in V$ labelled by an output transition of the process ($l(v_{max}) \in T_o$). The event v_{max} is called final event.
- the final event v_{max} of every maximal LPO $(V, <, l) \in \mathfrak{Lpo}(\mathbf{R})$ is a unique maximal event, i.e. it fulfills $v < v_{max}$ for every $v \in V \setminus \{v_{max}\}$.

In the running example of the business process of an insurance company, the only input transition is "Receive Claim" and the only output transition is "Complete Claim". In this case, the formulated requirement is fulfilled by our example composed run shown in Figure 2.

A further problem is that one subprocess may be invoked multiple times. Every occurrence of the refined transition in the superprocess initiates the subprocess.

The simplest solution for this problem is to synthesize the net of the subprocess from $\mathbf{R}^\star$ instead of $\mathbf{R}$. This exactly models that the behaviour given by $\mathbf{R}$ can occur arbitrarily often. Thus, if in the synthesized net the execution of one maximal run of $\mathfrak{Lpo}(\mathbf{R})$ is finished by an output transition, a state of the process is reached, in which the behaviour given by $\mathbf{R}$ (starting with an input transition) is again enabled, i.e. after one execution of the sub-process the initial marking (or some behaviourally equivalent marking) of the

subprocess is restored. Since the input transitions of the subprocess are connected to the input places of the refined transition, the behaviour of the supernet ensures that the input transitions are only enabled again, if the refined transition would be enabled again. Therefore, the subprocess still has to be enabled by the surrounding process. It can be invoked multiple times and each time the same behaviour is possible. Thus, the subprocess behaves as expected.

But also more intricate solutions are possible for the problem. First ingoing arcs from a place of the subprocess to an input transition of the subprocess can easily be omitted in the synthesis algorithm (respective variables are omitted). This ensures that each time the refined transition would be enabled, the initial transitions of the subprocess are enabled. Additionally, the marking of the subprocess after certain maximal runs of $\mathfrak{Lpo}(\mathbf{R})$ can be established in the synthesis algorithm (by modifying respective inequation systems [BDLM07]). In this approach, it is again possible to constitute that the marking reached after the execution of a maximal run coincides with the initial marking of the process. This is the standard case. But it is also possible to define other follower markings after one invocation of the subprocess and to vary the follower markings w.r.t. the actually executed maximal run. This models that the subprocess has a *memory*, which runs have actually been executed in prior invocations of the subprocess.

A memory can also be introduced by modelling several behaviours of the subprocess depending on the memory given by prior executed behaviour. That means the memory is modelled in the specification by the composed run $\mathbf{R}$. To enable this, the requirements formulated for $\mathbf{R}$ have to be changed. It has to be allowed that maximal runs of $\mathfrak{Lpo}(\mathbf{R})$ are a sequential composition of runs, which fulfill the properties required before, i.e. an event labelled by an output transition can directly be followed by an event labelled by an input transition. Thus, it can be specified for each execution of the subprocess ending with an output transition, which follower behaviour is then possible in the subprocess. Again the follower behaviour has to be invoked from the surrounding net.

An example of a subprocess with memory is shown in Figure 4. In the first invocation of the subprocess, only the sequence cd is executable, in a second invocation only ab is possible and in a third one again cd, and so on.

4 Case Study

In this section we briefly illustrate the proposed synthesis procedure by a small case study. We use our toolset VipTool, which has respective synthesis functionalities. The refinement aspects of the business process design procedure of [Des08] are not supported by VipTool. Thus we consider the generation of a main process to show the applicability of the synthesis algorithm. Since so far also the concept of composed runs is not implemented, we can only consider a specification given by a collection of alternative single runs, i.e. only the $+$ operator is supported. The alternative runs depicted in the screenshot of Figure 5 model the same behaviour as given by the composed run of Figure 2, except that not an arbitrary iteration of the activity "Ask Additional Queries" is possible, but the

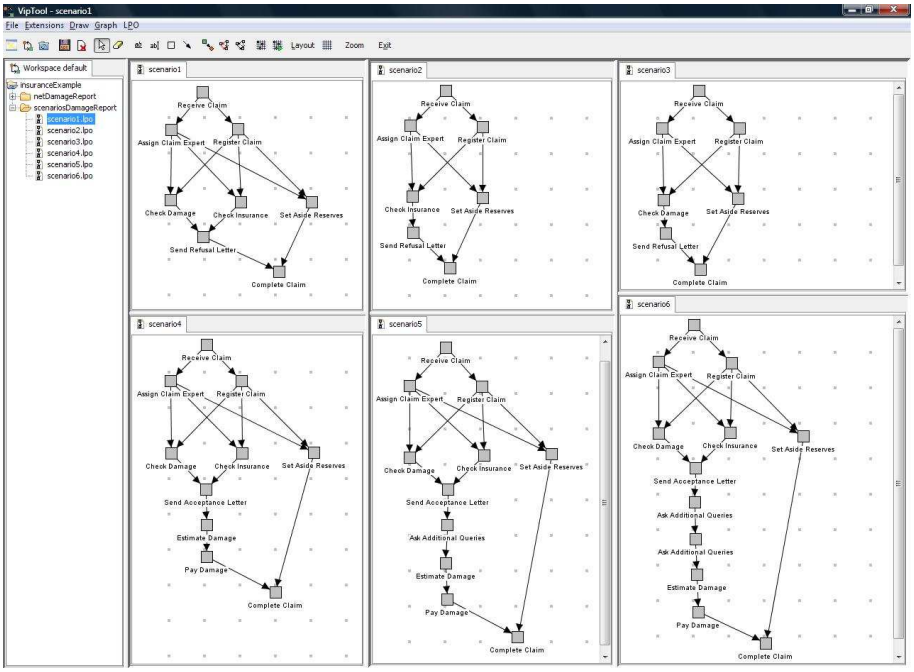

Figure 5: Screenshot of VipTool showing the user interface of the editor for partially ordered runs.

repetition is restricted to two times. This set of alternative runs forms the specification of the considered workflow of processing a claim in an insurance company.

There are six possible runs: All start by receiving a claim submitted by a client (activity ”Receive Claim”), followed by two concurrent activities ”Assign Claim Expert” and ”Register Claim”. The first one models the assignment of a claim expert in charge for this claim, the latter is concerned with the registration of the client and the loss form. Then concurrently reserves for the claim are established (activity ”Set Aside Reserves”) and the evaluation of the claim is started. The evaluation begins with two concurrent activities ”Check Damage” and ”Check Insurance”. ”Check Damage” represents checking validity of the clients insurance, ”Check Insurance” models checking of the damage itself. Scenario 4-6 model the situation that both checks are evaluated positively, meaning that an acceptance letter is sent (activity ”Send Acceptance Letter”) after the two checks. If one evaluation is negative, the company sends a refusal letter. Thus, the activity ”Send Refusal Letter” is performed after the two ”Check” activities if one is evaluated negatively (scenario 1). If a negative evaluation of one ”Check” activity already causes sending a refusal letter, while the other ”Check” activity has not been performed yet, this second ”Check” activity has to be disabled (i.e. it does not occur in a respective scenario), since it is no more necessary (scenario 2 and 3). In the case an acceptance letter is sent (scenario 4-6), either (scenario 4) the damage is immediately estimated (activity ”Estimate Damage”) and then payed (activity ”Pay Damage”), or additional queries to improve estimation of the loss (activity ”Ask Additional Queries”) are asked one (scenario 5) or two (scenario 6)

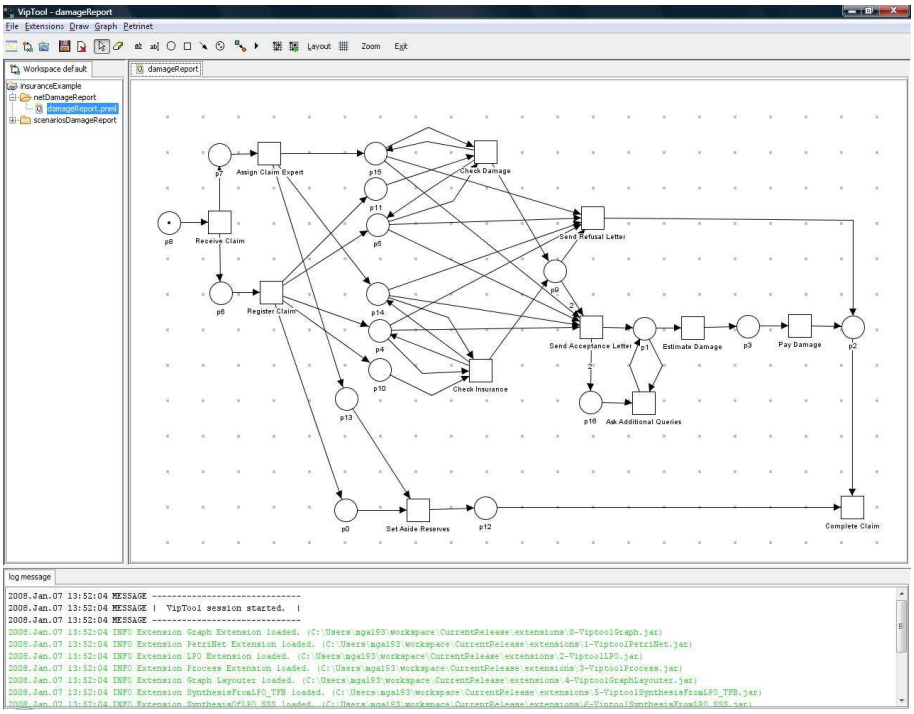

Figure 6: Screenshot of VipTool showing the user interface of the editor for Petri nets.

times before the damage is estimated. If the evaluation of the claim (including possibly paying the damage) and the activity ”Set Aside Reserves” are finished, the process can be completed by the activity ”Complete Claim”.

Figure 6 shows the net damageReport.pnml automatically created with the synthesis algorithm of VipTool from the specification depicted in Figure 5. The net represents a very compact model of the described complex business process. An algorithm of VipTool to check, whether the set of runs of the synthesized net damageReport.pnml coincides with the specified set of runs (scenarios 1-6) yields a positive answer (the synthesis algorithm only ensures a best upper approximation).

The example illustrates that directly designing a Petri net model of a business process is often challenging, while modelling runs and synthesizing a net is easy. Manually developing a complex Petri net such as the net damageReport.pnml for the described workflow is an error-prone task. But the design of the six runs 1-6 is straightforward, in particular if the concepts of scenario integration by composed runs are used (this supports that the single runs can be modelled by domain experts). A Petri net is automatically created from the specification by the synthesis algorithm.

5 Conclusion

In [Des08], general ideas for the application of Petri net synthesis in business process design were presented. In this paper, we showed a concrete procedure to design a process model using synthesis methods.

This workshop contribution constitutes a first proposal of the approach. The paper did not elaborate all aspects of the procedure in detail. A lot of work still has to be done. In particular methods tuning the synthesis algorithms to better practical applicability are important. This mainly concerns complexity and the size of the synthesized nets. Such methods are already discussed in the context of process mining [BDLM07, vdA07]. An advantage in our setting is that the presented refinement approach helps to keep the single processes small. Finally, results from practical application and evaluation will be important for further development. More precisely, the suggested novel approach to design business processes is based on several assumptions, but we believe that it could be helpful in many cases. However, this research is still in an initial phase and we do not have experiences from real applications. Future work includes defining of success criteria and empirical research. In particular, it would be interesting to identify and characterize settings in which our approach is superior to other approaches.

References

[BD96] E. Badouel and P. Darondeau. Theory of Regions. In *Petri Nets, LNCS 1491*, pages 529–586, 1996.

[BDJL06] R. Bergenthum, J. Desel, G. Juhás, and R. Lorenz. Can I Execute My Scenario in Your Net? VipTool Tells You! In *ICATPN 2006, LNCS 4024*, pages 381–390, 2006.

[BDLM07] R. Bergenthum, J. Desel, R. Lorenz, and S. Mauser. Process Mining Based on Regions of Languages. In *BPM 2007, LNCS 4714*, pages 375–383, 2007.

[BLM06] R. Bergenthum, R. Lorenz, and S. Mauser. Synthese von S/T-Netzen aus unendlichen partiellen Sprachen. In *AWPN 2006*, pages 1–8, 2006.

[BM07] R. Bergenthum and S. Mauser. Experimental Results on the Synthesis of Petri Nets from Partial Languages. In *Petri Net Newsletter*, volume 73, pages 3–10, 2007.

[Des99] J. Desel. Formaler Umgang mit semi-formalen Ablaufbeschreibungen. In *Angewandte Informatik und Formale Beschreibungsverfahren*, pages 45–90, 1999.

[Des08] J. Desel. From Human Knowledge to Process Models. In *Proceedings of UNISCON (to appear)*, 2008.

[DJLN03] J. Desel, G. Juhás, R. Lorenz, and C. Neumair. Modelling and Validation with VipTool. In *Business Process Management 2003, LNCS 2678*, pages 380–389, 2003.

[FMK03] G. Fliedl, H. C. Mayr, and C. Kop. From scenarios to KCPM dynamic schemas – aspects of automatic mapping. In *International Conference on Applications of Natural Language to Information Systems, LNI P-29*, pages 91–105, 2003.

[Gli95] M. Glinz. An Integrated Formal Model of Scenarios Based on Statecharts. In *ESEC 1995, LNCS 989*, pages 254–271, 1995.

[Gli00] M. Glinz. Improving the Quality of Requirements with Scenarios. In *Proceedings of the Second World Congress on Software Quality*, pages 55–60, 2000.

[LBDM07] R. Lorenz, R. Bergenthum, J. Desel, and S. Mauser. Synthesis of Petri Nets from Finite Partial Languages. In *ACSD 2007, IEEE*, pages 157–166, 2007.

[May98] H. C. Mayr. Towards business process modeling in KCPM. In *Petri nets and Business Process Management, Dagstuhl-Seminar-Report Nr. 217*, page 27. Springer, 1998.

[MK02] H. C. Mayr and C. Kop. A user centered approach to requirements modeling. In *Modellierung 2002, LNI P-12*, pages 75–86, 2002.

[MKE07] H. C. Mayr, C. Kop, and D. Esberger. Business process modeling and requirements modeling. In *First International Conference on the Digital Society (ICDS'07)*, pages 8–11, 2007.

[MS93] H. C. Mayr and M. Schnattler. Vorkonzeptuelle und konzeptuelle Dynamik-Modellierung. Petri-Netze im Einsatz für Entwurf und Entwicklung von Informationssystemen. In *Informatik aktuell*, pages 3–18. Springer, 1993.

[SMG06] C. Seybold, S. Meier, and M. Glinz. Scenario-Driven Modeling and Validation of Requirements Models. In *SCESM 2006, ACM*, pages 83–89, 2006.

[vdA07] W. M. P. van der Aalst. Finding structure in unstructured processes: the case for process mining. In *ACSD 2007, IEEE*, pages 3–12, 2007.

[vdADtH05] W.M.P. van der Aalst, M. Dumas, and A. H. M. ter Hofstede. *Process-Aware Information Systems – Bridging People and Software.* Wiley, 2005.

[vdAvH02] W.M.P. van der Aalst and K. van Hee. *Workflow Management: Models, Methods, and Systems.* MIT Press, Cambridge, Massachsetts, 2002.

[Wes07] M. Weske. *Business Process Management – Concepts, Languages and Architectures.* Springer, 2007.