

Towards Analyzing Declarative Workflows

Dirk Fahland*

Humboldt-Universität zu Berlin, Institut für Informatik,
Unter den Linden 6, 10099 Berlin, Germany
fahland@informatik.hu-berlin.de

Abstract. Enacting tasks in a workflow cannot always follow a pre-defined process model. In application domains like disaster management workflows are partially specified and circumstances of their enactment change. There exist various approaches for formal workflow models that are effective in such situations, like declarative specifications instead of operational models for formalizing flexible workflow process. These powerful models leave a gap to existing techniques in the domain of workflow modeling, workflow analysis, and workflow management.

In this paper we bridge this gap with a compositional mechanism for translating declarative workflow models to operational workflow models. The mechanism is of a general nature and we reveal its principles as we provide an exemplary definition for translating DecSerFlow models based on LTL to Petri nets. We then demonstrate its use in analyzing and refining declarative models.

1 Introduction

The classical notion of a *workflow* – the automation of a work procedure – relies on known and predictable circumstances within which it is enacted. A concrete enactment, a *case* of a workflow, is based on a fixed set of fully specified workflow processes, manipulating a certain kind of data and involving a known set of resources in a certain organizational structure [35]. There are application scenarios like disaster management where a case’s circumstances are only partly predictable. Established solutions are insufficient in such a situation.

Research in workflows treats these challenges as ‘change in workflow (management)’. Starting in known circumstances, any violation of these manifests itself as an observable change in the workflow’s environment that can be handled accordingly by changing the workflow as well.

The core of a workflow is the *process logic* that relates the involved tasks, resources and data to each other. Its formalization is the *workflow model* [35]. Most changes in the environment will cause a change in the process logic. The workflow model therefore must be able to reflect, support or incorporate changes in a well-defined way. This involves, among others, three different kinds of change: An *adaptive* process can change in order to react on exceptional circumstances like an unpredicted event, a *dynamic* process can transform and migrate its cases to

* The author’s work is funded by the DFG-Graduiertenkolleg 1324 “METRIK”.

conform with the changed process, and a *flexible* process can be executed despite having only incomplete process information [27].

Problem Description. As each kind of change in workflows is a challenge on its own, solutions and proposals for solutions use various formalisms [26, 32]; the desire to achieve a solution within a single formalism needs no justification.

Most of the works mentioned above address dynamic and/or adaptive workflow models using *Petri nets* [23] or graph-based formalizations with concepts that are similar to Petri nets. The approaches addressing flexible workflows, *Pockets of Flexibility* [26] and *DecSerFlow* [32], use declarative elements to formalize (parts of) a flexible process: Constraints on the occurrence of tasks reflect the work procedure’s minimal requirements on an execution. A concrete ordering of the tasks, as it must be given in a usual operational model, is determined at runtime and can be adjusted to the (changed) circumstances.

In contrast to established operational workflow models, predicting the behavior of (even just partly) declarative models is difficult by simple inspection. A property of interest would be the absence of deadlock and livelocks in the workflow, called *weak termination* [19]. It is therefore desirable to supplement these models with analysis techniques. Furthermore, none of the approaches above covers all aspects of change in a single formalism which makes it difficult to come up with a dynamic, adaptive, and flexible workflow model.

Contribution. In this paper, we propose a compositional, equivalent translation of declarative workflow models into operational workflow models. We have chosen Petri nets as our operational workflow model. Their simple concepts and formal semantics, and the powerful analysis and verification techniques for Petri net models make them a reasonable candidate [1]. By the compositionality of our transformation, analysis results of the transformed Petri net model can be related back to the original declarative model. Hence, we enable Petri net analysis techniques for declarative workflow models.

Additionally, our transformation makes a first step towards combining existing techniques for change in workflows as we may study the relationship between declarative and operational workflow models on the same terms.

The paper is organized as follows. In Section 2, we present the workflow models we are concerned with and provide an extended summary on existing works for change in workflows. We then show in Section 3 our approach of translating declarative to operational workflow models and provide the arguments for the correctness of the translation. By the help of a case study we show in Section 4 how the translation can be used to understand, analyze and refine declarative workflows. We conclude and discuss related works in Section 5.

2 Workflows Models and Change

2.1 Petri Net based Workflows

Petri nets [23, 29] are an established model for workflows [33]. We repeat the formal definitions and notations we need for this paper.

A *inhibitor reset Petri net* (IR net for short) $N = (P, T, F, I, R)$ extends the notion of classical Petri nets [9]: N consists of a finite set P of *places*, a finite set T of *transitions* ($P \cap T = \emptyset$), the *flow* relation $F \subseteq (T \times P) \cup (P \times T)$, $I \subseteq T \times P$ the set of *inhibitor arcs*, and $R \subseteq T \times P$ the set of *reset arcs*. In Petri net based workflows, tasks are modeled as transitions, resources are modeled as places and the arcs between them denote dependencies between tasks and resources.

For a *node* $x \in P \cup T$ its preset $\bullet x =_{\text{df}} \{y \mid (y, x) \in F\}$ and postset $x^\bullet =_{\text{df}} \{y \mid (x, y) \in F\}$ denote the direct predecessor and successor in F . The set $I(t)$ is the set of places tested by transition t ; set $R(t)$ is the set of places that is reset by t .

A *marking*¹ $m : P \rightarrow \mathbf{N}$ is a configuration of N , $m(p)$ denotes the number of *tokens* on p in m . p is *marked* in m if $m(p) > 0$. A transition $t \in T$ is *activated* (has *concession*) in m , iff $m \leq \bullet t$ and additionally $\forall p \in I(t) : m(p) = 0$. If t is activated, it may *fire* which results in the *successor marking* m''' , $m|t\rangle m'''$, computed as follows: $m' = m - \bullet t$; for all $p \in P$, $m''(p) = 0$ if $p \in R(t)$ and $m''(p) = m'(p)$ if $p \notin R(t)$; and $m''' = m'' + t^\bullet$.

A *marked IR net* $N = (P, T, F, I, R, m^0)$ has an *initial marking* $m^0 : P \rightarrow \mathbf{N}$. A *firing sequence* of N is a sequence $\langle t_1 t_2 t_3 \dots t_n \rangle$, $t_i \in T$ of transitions such that there exists a sequence of markings $m^1 m^2 m^3 \dots m^n$ of N with t_i enabled in m^{i-1} and $m^{i-1}|t_i\rangle m^i$ for all $i = 1, \dots, n$, denoted $m^0|t_1 \dots t_n\rangle m^n$. Let $R(N)$ denote set of all firing sequences of N . Petri net based workflows often have a unique initial place $\alpha \in P$ with $m^0 = [\alpha]$ and a set $\Omega \subseteq P$ of places that are marked iff the case has been handled completely [1, 17].

If $I = \emptyset$ and $R = \emptyset$, N is a *place transition Petri net* (P/T net for short) and the well-known equations may be derived. Petri nets have a graphical notation as bipartite, directed graphs with places drawn as circles and transitions drawn as boxes, edges are given by F . A black dot within a circle depicts a token on the respective place. In this paper, a double arrow between two nodes x and y depicts a loop $(x, y), (y, x) \in F$; see the nets in Fig. 1 and 2. Reset arcs are drawn as dashed lines, inhibitor arcs have a black dot at the transition (cf. Fig. 9).

2.2 Change in Workflows

Most of the classification schemes for ‘change in workflow (management)’ [31, 16, 27, 25] go along the same lines with emphasis on different aspects. A rather complete classification scheme is given in [31]. The authors propose six dimensions of change: reason for change, effect of change, affected workflow perspective (e.g. process or organization), kind of change, allowed moment of change, and treatment of existing cases under change. The classification is supplemented with a consideration of errors that may arise under change and how change can be dealt with.

The aspect we are mostly interest in is change in the process perspective of a workflow. The classification scheme of Heinel et al [16] did a first discrimination between loosely specified models and models that must be changed. The

¹ We conceive a marking as a *multiset* and canonically extend \leq , $+$ and $-$ on \mathbf{N} to markings. Ordinary sets of places are conceived as special multisets.

classification of Sadiq et al [27] extends on this and names three dimensions to describe the capability of a workflow for behaving under varying circumstances:

- *Dynamism*: Change of the workflow due to evolution of the real-world process it is implementing and correct transformation of the workflow’s cases.
- *Adaptability*: Change of the workflow due to exceptional circumstances like unexpected events or faults.
- *Flexibility*: Capability of correctly executing an underspecified workflow to guarantee correct runs in various or changing environments of the workflow.

Dynamism and Adaptability are properties that actually relate two different workflows to each other: the original workflow that encounters a change in the environment and the changed workflow that is compatible with the ‘new’ environment. Flexibility aims on covering many different feasible runs in a humanly-conceivable model.

Each of these dimensions is significant for processes in disaster management: (1) The actual work procedures as they are performed in disaster management change for various reasons, like gained experience, change in administrative directives etc; any workflow system supporting these procedures has to evolve. (2) Exceptional circumstances for a given workflow are the usual circumstances in case of a disaster; a feasible workflow system must deal with unpredicted events and changes. (3) Because unpredictability is inherent in disasters, work procedures that are effective in such situations are loosely or partially specified to provide the necessary degree of freedom for the change.

It can be seen from the definitions in the previous section that classical Petri net based workflows lack in supporting change in the process perspective of a workflow in first place. In the following we focus on flexibility of workflows. But we assume that neither of the above dimensions can be treated in isolation if we are interested in dynamic, adaptive *and* flexible workflows. We therefore give a brief overview on existing works in that field.

Dynamic and Adaptive Workflows A survey by Rinderle et al [24, 25] presents a number of existing approaches for dynamic and adaptive workflows. All approaches mentioned there (Petri net based workflow nets [30], flow nets [11], workflow net model/workflow sequential model [2], WIDE [5, 6], WASA₂ [34], Adept [22]) follow a graph-based approach to denote a workflow’s tasks and their ordering. Changes are operations on the graph structure. Their mutual similarity is not coincidental as graph based approaches are widely used to make the procedural knowledge of work procedures explicit. Despite being capable of modeling and sometimes analyzing dynamic and adaptive workflows, each of these approaches assumes a complete model of the workflow. Unfortunately, in some application domains like disaster management, such a complete model cannot be derived as hardly any two executions are the same: existing graph-based approaches hardly support flexible workflows.

Flexible Workflows Achieving flexibility of a workflow requires a concise representation of many behaviors. A simple example is the arbitrary but sequential

execution of n tasks [27]. An explicit representation of such a workflow gives rise to $n!$ possible sequences among which one can choose and switch in between. This exploding representation does not even meet the intention of the requirement. Finding adequate means for modeling and enacting such behavior is the challenge. The subsequent sections will present two approaches for flexible workflows.

2.3 Pockets of Flexibility

Sadiq, Sadiq and Orlowska [26, 27] achieve a compact notion of many behaviors through an intended underspecification of parts of the process. Confined to the workflow paradigm, a flexible workflow is given by tasks and their ordering in a graph-based model. Flexibility is introduced by the concept of a ‘build’ task. Each ‘build’ task contains a number of workflow fragments. The ordering of these workflow fragments is left open at design-time; constraints provided by the modeler restrict the set of valid compositions.

The workflow fragments of a ‘build’ task are brought into a partial order at runtime when that ‘build’ task is executed. Because the resulting composition may differ in each execution a ‘build’ task establishes a *pocket of flexibility* in the workflow specification. Thus the key idea of this approach is to build a complete (graph-based) workflow model as the workflow is being executed.

The semantics of their workflow model allows for verifying some properties, like conflict-freeness, transitivity and redundancy of constraints. A corresponding engine to execute this kind of workflows has been implemented by Sadiq et al. The authors also provide ideas how their approach can be extended towards adaptivity and dynamism. [27]

2.4 Declarative Service Flow Language

Compared to the ‘pockets of flexibility’, Aalst and Pesic [32] went a step further by modeling a workflow’s behavior entirely in temporal-logic constraints over its tasks. The constraints are formalized in a subset of linear-time temporal logic (LTL). Any sequential ordering of finitely many, (possibly repeated) occurrences of the workflow’s tasks (i.e. any path) that satisfies such an LTL formula is a run of the workflow. In this sense, anything may happen as long as the final run satisfies all constraints. The idea is to let the modeler specify the constraints of the process to be supported rather than a solution in operational semantics. We briefly remember syntax and semantics of LTL and then present its application to specify workflow behavior.

Linear-Time Temporal Logic (LTL) LTL is a modal logic that allows to specify the future of paths in terms of a set $Prop$ of atomic propositions. The set $\mathcal{F}_{LTL}(Prop)$ of LTL formulas is the least set containing the atomic proposition $Prop$ and that is closed under boolean connectives, the unary temporal operators \bigcirc (next), \square (always), \diamond (eventually) and the binary operator \mathcal{U} (until). [21, 12]

A *path*, is an infinite sequence of interpretations of the propositions: $r : \mathbf{N} \rightarrow 2^{Prop}$; each position $i \in \mathbf{N}$ of r is conceived as a *state*. An LTL formula $\phi \in \mathcal{F}_{LTL}(Prop)$ is evaluated at a state $i \in \mathbf{N}$ of a path r :

$$\begin{aligned} r, i &\models p \in Prop \text{ iff } p \in r(i), \\ r, i &\models \bigcirc \phi \quad \text{iff } r, i + 1 \models \phi, \\ r, i &\models \square \phi \quad \text{iff } \forall j \geq i : r, j \models \phi, \\ r, i &\models \diamond \phi \quad \text{iff } \exists k \geq i : r, k \models \phi, \\ r, i &\models [\phi \mathcal{U} \phi'] \text{ iff } \exists k \geq i : r, k \models \phi' \wedge \forall i \leq j < k : r, j \models \phi, \end{aligned}$$

and boolean connectives are evaluated as usual. Path r is a *model* for $\phi \in \mathcal{F}_{LTL}(Prop)$, $r \models \phi$, iff $r, 0 \models \phi$. For a finite prefix $r|_{\{1, \dots, n\}}$ of a path, the definition of \models can be adapted by restricting states and quantifiers to $\{1, \dots, n\}$.

LTL for Workflows Aalst and Pesic have chosen a special subset of LTL to support workflows. A *declarative* workflow $D = (Tasks_D, \phi_D)$ has a set $Tasks_D$ of atomic tasks that may be enacted in D . The LTL formula $\phi_D \in \mathcal{F}_{LTL}(Prop(D))$ restricts the occurrences of these tasks. For this purpose, the set of atomic propositions $Prop(D)$ has a special shape: $Prop(D) =_{\text{df}} \{activity = A \mid A \in Tasks_D\}$. If $activity = A$ holds in a state, the interpretation is, that this state is left by enacting task A . Therefore, a path $r : \mathbf{N} \rightarrow 2^{Prop(D)}$ is *feasible* for D if $\forall i : |r(i)| \leq 1$. The behavior that is specified by D is the set of feasible runs satisfying ϕ_D , $R(D) =_{\text{df}} \{r \models \phi_D \mid r \text{ is feasible}\}$.

The nature of $\mathcal{F}_{LTL}(Prop(D))$ is still too general to derive properties for $R(D)$ that are useful in the workflow domain. Aalst and Pesic defined a set \mathcal{F}_{WF} of parameterized LTL constraint templates as the basic building blocks of the *declarative service flow language* (DecSerFlow) [32]. By instantiating the templates, one obtains a set $\mathcal{F}_{WF}(D) \subseteq \mathcal{F}_{LTL}(Prop(D))$ of LTL *workflow constraints*. A *DecSerFlow workflow* (DSF workflow for short) is a declarative workflow D where $\phi_D = \bigwedge_{i=1}^n \phi_i$ is a conjunction of workflow constraints, $\phi_i \in \mathcal{F}_{WF}(D), i = 1, \dots, n$.

The constraints include the existence, absence, or counted occurrence of a task in a run; positive constraints on the occurrence of two tasks like ‘task A requires a preceding B ’; and negative constraints like ‘ A and B are mutually exclusive’. Each constraint comes with a graphical representation which allows a declarative specification without looking into LTL formulas. Using LTL as their model, adding further templates to \mathcal{F}_{WF} is feasible.

To enact a DSF workflow D , the authors suggest the translation of the LTL formula ϕ_D to a Büchi automaton [7] which serves as the basis for the workflow in the corresponding engine. This approach is well studied and yields an equivalent operational model for each declarative workflow [13]. The setting of infinite paths for LTL and Büchi automata can be tailored to the setting of finite paths for workflows on a technical level, see [32] for the details. The authors mention that, despite these results, “the automatic construction of an automaton suitable for enactment and on-the-fly monitoring is far from trivial” [32, chap.5].

Relation to ‘Pockets of Flexibility’ The principle idea for achieving flexibility in DecSerFlow and in the model of Sadiq et al is the ordering of a set of unordered tasks under constraints at runtime. The constraints in either approach are similar or even the same. We therefore think, without wanting to prove this here, that a ‘build’ task can be conceived as a DSF workflow within an operational model. For this reason, and because DecSerFlow has with LTL a more general and formally well-defined model, we take DecSerFlow as the basis for our translation.

3 Translating Declarative Workflow to Operational Workflows

In this section we present our idea for translating declarative workflows to operational workflows. As justified in Sections 1 and 2.4, we have chosen DecSerFlow as our declarative model while the operational model will be Petri nets. This translation does not require any properties that are inherent to these formalisms; they serve as mere representatives for either domain. The aim of this section is to reveal the basic ideas that are involved in such a translation rather than a complete translation of DecSerFlow.

We proceed as follows. We present some of the DecSerFlow constraint templates and explain our approach for a translation into Petri nets. The choice for the templates to be presented here is guided by our case study in Section 4. We explain the decisive arguments for correctness as we proceed in the presentation.

3.1 Approach for Translating

For the remainder of this paper, let $D = (Tasks_D, \phi_D)$ be a DSF workflow. The aim of our translation is to obtain a Petri net $N(D)$ such that every firing sequence of $N(D)$ can be mapped to a run of ϕ_D and vice versa.

Our translation approach has been exercised before, e.g [17]; it is intensional: for each construct of DecSerFlow we analyze its effects on the paths of the system. We then search for a Petri net construct that achieves the same effect. Our translation makes no use of the inner structure of LTL formulas, but their formal foundation makes them extensionally well-defined. We translate each of DecSerFlow’s language concepts to a Petri net pattern or an operation on Petri nets as follows:

- The notion of a task $A \in Tasks_D$ is translated into a dedicated pattern,
- we define a pattern to model the acceptance of finite paths, and
- we define a parameterized Petri net pattern $N(\psi)$ for each of the parameterized constraint templates $\psi \in \mathcal{F}_{WF}$.
- Instantiating parameters \mathbf{x} to values \mathbf{v} of a constraint template ψ to a constraint $\phi = \psi[\mathbf{x} \mapsto \mathbf{v}] \in \mathcal{F}_{WF}(D)$ is translated as instantiation of the Petri net pattern $N(\psi)$ to a *constraint net* $N(\phi) = N(\psi)[\mathbf{x} \mapsto \mathbf{v}]$.
- A conjunction of two constraints is translated as ‘merging’ of the corresponding constraint nets, which will be formally the union of the nets.

The intriguing part in this approach is a mismatch between models of an LTL formula and firing sequences of a Petri net which we will discuss first before going into details.

Being an LTL formula, the model of a DecSerFlow constraint $\phi \in \mathcal{F}_{WF}(D)$ is the set $R(\phi)$ of all infinite paths that satisfy ϕ . Paths in $R(\phi)$ also make statements about occurrences of activities that are not mentioned in ϕ as $R(\phi)$ is built using all atomic propositions in $Prop(D) = \{activity = A \mid A \in Tasks_D\}$. The firing sequences in $R(N)$ make statements about transition of N only. We solve this mismatch by examining the DSF workflow $D_\top =_{df} (Tasks_D, \top)$ that does not constrain the occurrences of its tasks. $R(D_\top)$ is the space $(\mathbf{N} \rightarrow 2^{Prop(D)})$. The Petri net that exhibits the same behavior contains a transition $task_A$ for every task $A \in Task(D)$ and a marked place pre_A that is in the pre- and post-set of t_A , see net N_A in Fig. 1. This net will be the canvas for our construction.

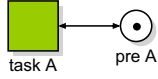


Fig. 1. Basic Petri net pattern N_A for the occurrence of task A , parameter: A .

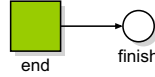


Fig. 2. Petri net pattern N_{end} to denote the end of a run.

When modeling workflows, we are interested in finite executions only. More precisely, after finitely many steps, it must be possible to decide whether a path or the firing sequence is a model for ϕ_D . We only then may stop enacting activities. In a Petri net model satisfiability is checked operationally via activating or not activating a transition. We have chosen the end transition of net N_{end} in Fig. 2 to perform this check: our construction will introduce pre-places of end that are marked only if a corresponding constraint has been satisfied. A token on end 's post-place $finish$ denotes an accepting state where all constraints have been satisfied. We include the pattern in our canvas $N_D(\top)$:

$$N_D(\top) =_{df} \bigcup_{act \in Tasks(D)} N_A[A \mapsto act] \cup N_{end} \quad (1)$$

Observe that the firing sequences of $N_D(\top)$ correspond to the runs of D_\top .

We can now formalize the translation procedure from D to $N(D)$. *Merging* two IR Petri nets N_1 and N_2 yields the IR Petri net

$$N_1 \bowtie N_2 =_{df} (P_1 \cup P_2, T_1 \cup T_2, F_1 \cup F_2, I_1 \cup I_2, R_1 \cup R_2, m_1^0 + m_2^0). \quad (2)$$

\bowtie is associative and commutative. Merging $N_D(\top)$ with a constraint net that contains transitions and places of $N_D(\top)$ reduces the set of firing sequences just as conjoining \top with a constraint on $Task(D)$ reduces the set of satisfied runs.

The DSF workflow $D = (Tasks_D, \phi_D)$ can be conceived as the conjunction of instantiated constraint templates: $\phi_D = \bigwedge_{i=1}^n \phi_i$ with $\phi_i = \psi_i[\mathbf{x}_i \mapsto \mathbf{v}_i]$, $\psi_i \in \mathcal{F}_{WF}$, $i = 1, \dots, n$.

We have chosen to generate the Petri net workflow $N(D)$ from this inner structure of D . Assume a Petri net $N(\psi_i)$ for each $\psi_i \in \mathcal{F}_{WF}$. For $i = 1, \dots, n$ we define $\xi_i =_{df} \xi_{i-1} \wedge \phi_i$, with $\xi_0 =_{df} \top$, and

$$N_D(\xi_i) =_{df} N_D(\xi_{i-1}) \bowtie N(\psi_i)[\mathbf{x}_i \mapsto \mathbf{v}_i]. \quad (3)$$

We define $N(D) =_{df} N_D(\phi_D) = N_D(\xi_n)$ to be the result of our translation. An *accepting* firing sequence of $N(D)$ leads to a marking m with $m(\text{finish}) > 0$. The translation yields an *equivalent* net $N(D)$ iff every path of D can be mapped to an accepting firing sequence of $N(D)$ and vice versa.

The design challenge for the constraint nets $N(\psi_i)$ is to chose the net's elements in such a way that the merge operation does restrict the behavior only wrt ψ_i . We exemplary demonstrate the ideas for constructing equivalent constraint nets for two constraint templates.

3.2 Constructing Constraint Nets

DecSerFlow contains a set of constraints to restrict the number of occurrences of a task in a run [32]. The most basic constraint requires that a task A has to occur in a run: $\text{ex}(A) =_{df} (\diamond \text{activity} = A)$. By the help of the ‘next’ operator \bigcirc one can count occurrences of tasks. The formula $\diamond (\text{activity} = A \wedge \bigcirc \text{ex}(A))$ evaluates to *true* if task A has occurred twice (or more often). Its negation evaluates to *true* if A has occurred at most once. In this style, DecSerFlow provides a parameterized constraint template $\text{ex}_i^j(A)$ that evaluates to *true* if task A has occurred at least i times and at most j times.

The corresponding constraint net is depicted in Figure 3. Firing of transition task_A is bounded by the number i of tokens in its pre-place A_{\max} . For each firing of task_A , a token on its post-place A_{executed} is produced. Thus transition end is enabled only if task_A has fired at least j times.

In order to guarantee that task_A can fire at most i times, merging with any other instantiated constraint net must not result in a net that consumes or produces token on A_{\max} . All other constraint nets have to be defined in such a way. Otherwise, a constraint net $N(\psi)$ would not restrict the behavior of the system only wrt. the template $\psi \in \mathcal{F}_{WF}$.

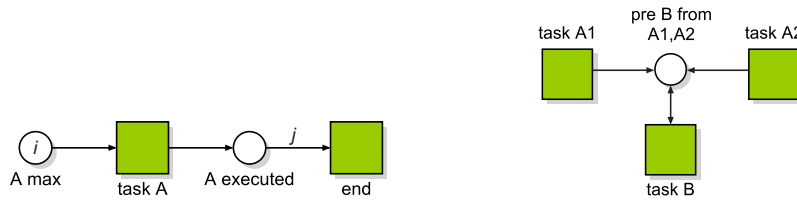


Fig. 3. Petri net pattern $N(\text{ex}_i^j(A))$; task A occurs at least i times and at most j times.

Fig. 4. Petri net pattern $N(\text{precedence}_2(\{A_1, A_2\}, B))$; task B is preceded by task A_1 or task A_2 .

DecSerFlow provides a number of constraint templates to order the occurrences of two (or more) different tasks. Our second example constraint requires that before any execution of task B , task A has to occur: $\text{precedence}(A, B) =_{\text{df}} \text{ex}(B) \rightarrow [(\neg \text{activity} = B) \cup \text{activity} = A]$. This can be generalized to an alternative precedence of two or more tasks: $\text{precedence}_k(\{A_1, \dots, A_k\}, B) =_{\text{df}} \text{ex}(B) \rightarrow [(\neg \text{activity} = B) \cup \bigvee_{i=1}^k \text{activity} = A_i]$

The constraint net for the generalized precedence (with $k = 2$) is depicted in Fig. 4. Transition task_B is prevented from firing through adding an empty pre-place $\text{pre}_B^{A_1, A_2}$. By making $\text{pre}_B^{A_1, A_2}$ post-place of task_{A_1} and task_{A_2} , their firing enables task_B . Observe that firing task_B does not remove the token from $\text{pre}_B^{A_1, A_2}$; this is necessary to realize all runs of $\text{precedence}_2(\{A_1, A_2\}, B)$ in the constraint net.

Consider the following DSF workflow $D_3 = (\{\text{receive}, \text{hotel}, \text{airline}\}, \phi_3)$ with

$$\begin{aligned} \phi_3 = & \text{ex}_1^1(\text{receive}) \wedge \text{ex}_1(\text{hotel}) \\ & \wedge \text{precedence}(\text{receive}, \text{hotel}) \wedge \text{precedence}(\text{receive}, \text{airline}) \end{aligned} \quad (4)$$

By instantiating and merging the patterns of Figures 3 and 4 according to (3) on our canvas $N_{D_3}(\top)$ we obtain the Petri net $N(D_3)$ depicted in Fig. 5.

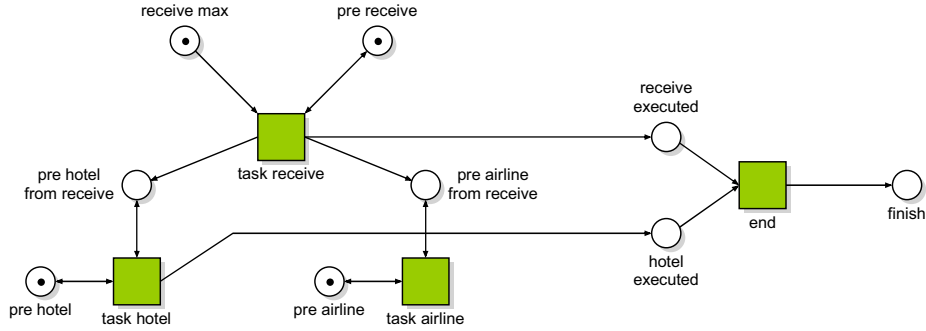


Fig. 5. Petri net $N(D_3)$ corresponding to specification (4) generated by instantiating and merging the patterns of Fig. 3 and Fig. 4 according to (3).

Observe that the firing sequence $\langle \text{task}_{\text{receive}}, \text{task}_{\text{hotel}}, \text{end} \rangle$ reaches an accepting state in which every constraint is satisfied. Further, from this state on $\text{task}_{\text{airline}}$ and $\text{task}_{\text{hotel}}$ may fire infinitely often. This firing does not violate any of the constraints and is therefore a run of D_3 . The simplicity of our construction comes at the price of an unbounded net.

3.3 More Complex Patterns

We will now briefly present the remaining constraint templates that we want to use in this paper and their corresponding constraint nets.

The constraint $\text{notResponse}(A, B)$ disallows any occurrence of B after an occurrence of A . Formally $\text{notResponse}(A, B) = \Box (activity = A \rightarrow (\Box \neg activity = B))$. The net $N(\text{notResponse}(A, B))$ is depicted in Fig. 6; the firing of task_B is prohibited by the inhibitor arc once the post-place of task_A is marked.

A more generalized variant of notResponse requires that tasks A and B must not occur in the same run. Its formal definition in LTL is $\text{notCoExistence}(A, B) =_{df} ((\Diamond activity = A) \rightarrow (\Box \neg activity = B)) \wedge ((\Diamond activity = B) \rightarrow (\Box \neg activity = A))$. The firing sequences of $N(\text{notCoExistence}(A, B))$ in Fig. 7 are correct and complete wrt this property: Firing of task_A produces a token on executed_A^B which prohibits the firing of task_B because of the inhibitor arc. Similarly for task_B . Place $\text{serialize}_{A,B}$ prevents the simultaneous firing of both transitions.

The similarity of $N(\text{notCoExistence}(A, B))$ and $N(\text{notResponse}(A, B))$ is inevitable: $\text{notCoExistence}(A, B)$ is extensionally equivalent to $\text{notResponse}(A, B) \wedge \text{notResponse}(B, A)$.

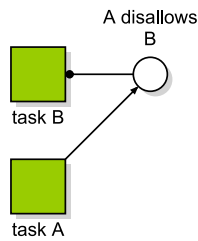


Fig. 6. Petri net pattern $N(\text{notResponse}(A, B))$

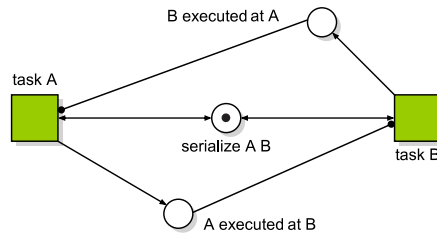


Fig. 7. Petri net pattern $N(\text{notCoExistence}(A, B))$

To require that in every run task A or task B (or both) have to occur, use $\text{mutualSubst}(A, B) =_{df} \Diamond (activity = A \vee activity = B)$. Fig. 8 shows the Petri net pattern where firing of end is subject to a preceding firing of task_A or task_B .

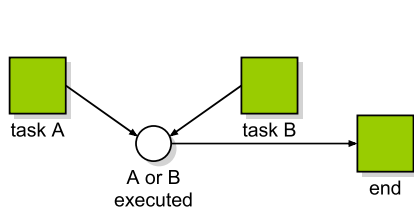


Fig. 8. Petri net pattern $N(\text{mutualSubst}(A, B))$

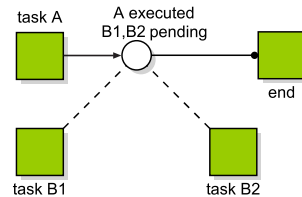


Fig. 9. Petri net pattern $N(\text{response}_2(A, \{B1, B2\}))$

A quite powerful, but in its effects underestimateable, constraint specifies that after the execution of task A one more more tasks B_1, \dots, B_k have to occur:

$\text{response}_k(A, \{B_1, \dots, B_k\}) =_{\text{df}} \square (\text{activity} = A \rightarrow (\diamond \bigvee_{i=1}^k \text{activity} = B_i))$. In Fig. 9 we depict the corresponding net $N(\text{response}_2(A, \{B_1, B_2\}))$. Firing task_A creates the commitment that its post-place has to be cleared from any tokens before end may fire to reach an accepting state. Because of the reset arc from either of the task_{B_i} to that place, firing any of task_{B_i} fulfils that commitment.

3.4 P/T Patterns

The patterns presented in the previous sections are a rather straight operational model for the DecSerFlow constraint templates. Some patterns require the more advanced Petri net concepts of inhibitor and reset-arcs (see Figures 6, 7, 9). The use of these arcs may turn into a disadvantage if it comes to analyzing the constructed Petri net. Most of the analysis techniques for Petri nets are defined for plain P/T nets (e.g. [23, 29, 28]). Translating IR-nets to P/T nets would not be an issue if our nets were bounded [29]; unfortunately our constraint nets do not enjoy this property.

We therefore provide a P/T net pattern $N'(\psi)$ that is branching bisimilar to the IR net pattern $N(\psi)$ for each constraint template $\psi \in \mathcal{F}_{\text{WF}}$ if any newly added transition in the P/T net is made invisible.

Two Petri nets N_1 and N_2 are *branching bisimilar* by the symmetric relation $\sim \subseteq \mathcal{M}(P_1) \times \mathcal{M}(P_2)$ iff $m_1^0 \sim m_2^0$ and for every pair of markings $m_1 \sim m_2$ and $m_1 | t | m'_1$ implies that either t is invisible and $m'_1 \sim m_2$ or there exists $m'_2 \in \mathcal{M}(P_2)$ with $m_2 | t_1 \dots t_k | m'_2$, $t_1 \dots t_k \in T_2$ invisible, and $m'_1 \sim m'_2$. [20]

Subsequently, we present the P/T patterns for `notResponse`, `notCoExistence`, and `response` together with the proof arguments for branching bisimilarity to the IR patterns.

notResponse. To realize `notResponse(A, B)` in a P/T net, the firing of task_A must imply that task_B has no concession anymore. Because $N'(\text{notResponse}(A, B))$ must not bound the number of occurrences of task_A , the concession cannot be removed by consuming a token from $\bullet \text{task}_B$ when task_A fires. We therefore have chosen to add an invisible transition choose_A that implements the choice to execute A . It removes the concession for task_B and gives it to task_A as soon as all other preconditions for firing task_A are met. The constraint net pattern $N'(\text{notResponse}(A, B))$ is depicted in Fig. 10.

The place pre_A^* is a parameter by itself and stands for the set $\{\text{pre}_A^x \mid \exists y : \text{pre}_A^y \in \bullet \text{task}_A \wedge y = x\}$ of places in the final composed net (for a chosen valuation of A). This set will contain any unmarked place in $\bullet \text{task}_A$ that needs to be marked to enable task_A , cf. the IR net patterns in the previous sections. By making the firing of choose_A dependent on this set of places, we guarantee that task_B loses its concession in the final P/T net iff it loses its concession in the final IR net. Enabling task_A depends on an invisible transition only. Hence replacing $N(\text{notResponse}(A, B))$ by $N'(\text{notResponse}(A, B))$ in the construction process results in a branching bisimilar net. Using pre_A^* requires a final step in the construction of the Petri net workflow, we call *resolving*, after all constraint

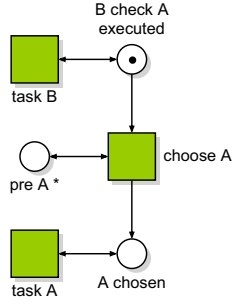


Fig. 10. P/T net:
 $N'(\text{notResponse}(A, B))$

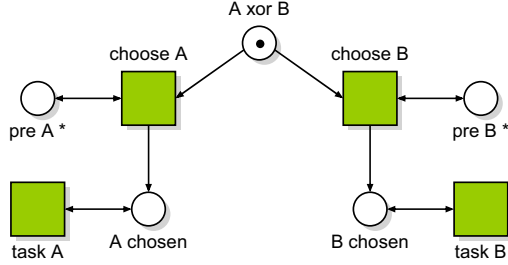


Fig. 11. P/T pattern: $\text{notCoExistence}(A, B)$

nets have been merged. The procedure will be formalized subsequently; for the moment, Fig. 12 illustrates the matter.

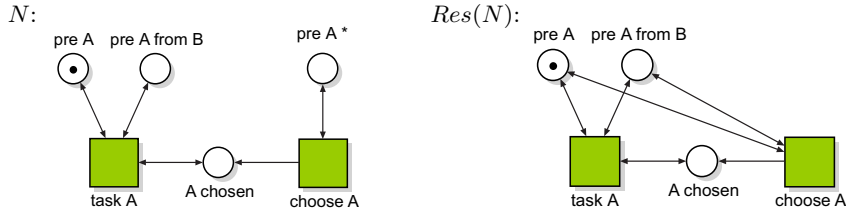


Fig. 12. Resolving place pre_A^* in net N yields the net $\text{Res}(N)$.

In $N'(\text{notResponse}(A, B))$, the transition choose_A is not parameterized with the name of task B . This is necessary as the choice to execute A has to be made globally. A set of local conflicting choices could otherwise lead to a deadlock if not each local choice is solved in favor of the same transition. This phenomenon is called *non-local choice* [15]. Any other pattern that ‘decides’ to fire A before A does fire has to use transition choose_A to solve any such conflict; our merge operator \bowtie then makes the choice global.

notCoExistence. As mentioned in Sect. 3.3, the constraint $\text{notCoExistence}(A, B)$ is extensionally equivalent to $\text{notResponse}(A, B) \wedge \text{notResponse}(B, A)$. The net $N'(\text{notCoExistence}(A, B))$ therefore uses the same constructs as $N'(\text{notResponse}(A, B))$ to prevent firing of task_B as soon as pre_A^* is marked and vice versa, cf. Fig. 11. Because occurrences of task_A and task_B are mutually exclusive according to the LTL constraint, the choice for either task is made nonreversible by removing the token from place xor_A^B . Branching bisimilarity to the IR pattern $N(\text{notCoExistence}(A, B))$ follows.

response. The constraint $\text{response}_k(A, \{B_1, \dots, B_k\}) =_{\text{df}} \square (activity = A \rightarrow (\diamond \bigvee_{i=1}^k activity = B_i))$ has proven to be particularly difficult to implement in a P/T net. The intriguing part is that *each* occurrence of A requires a subsequent occurrence of a B_i . The necessary bookkeeping in the IR net was facilitated by the reset arc. The P/T net solution has to differentiate between: (a) neither A nor B_i has fired, (b) A has fired, B_i has not yet fired, (c) A has fired and subsequently B_i has fired, and (d) B_i has fired without a (directly) preceding A . These cases are encoded by markings of $N'(\text{response}_2(A, \{B_1, B_2\}))$ in Fig. 13 as follows.

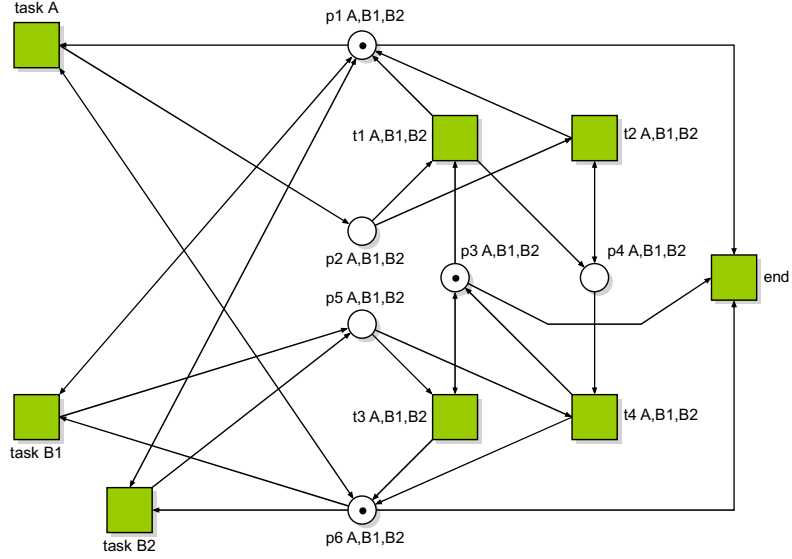


Fig. 13. P/T pattern: $\text{response}(A, \{B_1, B_2\})$.

- case (a): marking $m_1 = [p_1^{A, B_1, B_2}, p_3^{A, B_1, B_2}, p_6^{A, B_1, B_2}]$,
- case (b): marking $m_2 = [p_2^{A, B_1, B_2}, p_3^{A, B_1, B_2}, p_6^{A, B_1, B_2}]$
or $m'_2 = [p_1^{A, B_1, B_2}, p_4^{A, B_1, B_2}, p_6^{A, B_1, B_2}]$
or $m''_2 = [p_2^{A, B_1, B_2}, p_4^{A, B_1, B_2}, p_6^{A, B_1, B_2}]$,
- case (c): marking $m_3 = [p_1^{A, B_1, B_2}, p_4^{A, B_1, B_2}, p_5^{A, B_1, B_2}]$,
- case (d): marking $m_4 = [p_1^{A, B_1, B_2}, p_3^{A, B_1, B_2}, p_5^{A, B_1, B_2}]$.

Firing task_A in m_1 leads to m_2 and m'_2 , any subsequent firing of task_A in m'_2 leads to m''_2 and back to m'_2 . Firing task_{B_i} in m'_2 leads to m_3 and then to m_1 ; firing task_{B_i} in m_1 leads to m_4 and m_1 . Hence the interpretation of the markings is sound. task_A is disabled in m_3 and m_4 , task_{B_i} is disabled in m_2 and m''_2 . This guarantees that in $N'(\text{response}_2(A, \{B_1, B_2\}))$, only the markings m_1, \dots, m_4 are reachable. Finally, end is enabled in m_1 only, hence the accepting

state is reachable only if the constraint is satisfied. Thus the pattern is correct. Since occurrences of the tasks are not restricted, $N'(\text{response}_2(A, \{B_1, B_2\}))$ is branching bisimilar to $N(\text{response}_2(A, \{B_1, B_2\}))$.

Resolving pre_A^ in P/T nets.* The P/T net patterns $N'(\text{notResponse}(A, B))$ and $N'(\text{notCoExistence}(A, B))$, defined in the previous section, contain the parameterized place pre_A^* that needs to be *resolved* to the set $\text{Res}(\text{pre}_A^*) =_{\text{df}} \{\text{pre}_A^x \mid \exists x : \text{pre}_A^x \in \bullet \text{task}_A\}$ in the resulting Petri net workflow $N(\phi_D)$. Resolving means that each pre-transition (and each post-transition) of the place pre_A^* becomes a pre-transition (post-transition, respectively) of $p \in \text{Res}(\text{pre}_A^*)$; finally pre_A^* is removed from the resolved net.

We lift the *Res* operator to the level of P/T net $N = (P, T, F, m)$ with $\text{Pre}(N) = \{\text{pre}_A^* \in P \mid \text{task}_A \in T\}$:

- $\text{Res}(P) =_{\text{df}} P \setminus \text{Pre}(N)$,
- $\text{Res}(T) =_{\text{df}} T$,
- $\text{Res}(F) =_{\text{df}} \{(x, y) \in F \mid x, y \in \text{Res}(P) \cup \text{Res}(T)\}$
 $\cup \{(p, t) \mid (\text{pre}_A^*, t) \in F, \text{pre}_A^* \in \text{Pre}(N), p \in \text{Res}(\text{pre}_A^*)\}$
 $\cup \{(t, p) \mid (t, \text{pre}_A^*) \in F, \text{pre}_A^* \in \text{Pre}(N), p \in \text{Res}(\text{pre}_A^*)\}$,
- $\text{Res}(m)(p) =_{\text{df}} m(p)$ if $p \in \text{Res}(p)$ and $\text{Res}(m)(p) =_{\text{df}} 0$ otherwise,
- $\text{Res}(N) =_{\text{df}} (\text{Res}(P), \text{Res}(T), \text{Res}(F), \text{Res}(m))$.

The set of P/T nets is closed under $\text{Res}(\cdot)$, and $\text{Res}(\cdot)$ is idempotent for P/T nets.

Translating DSF workflows to P/T Nets. With the new P/T net patterns, we need to adapt our translation (3). Let $N'(\psi) =_{\text{df}} N(\psi)$ for each constraint template $\psi \in \mathcal{F}_{\text{WF}}$ for which we have not defined $N'(\psi)$ yet. Given $D = (\text{Tasks}_D, \phi_D)$, structured as in Sect. 3.1 by $\phi_D = \bigwedge_{i=1}^n \psi_i[\mathbf{x}_i \mapsto \mathbf{v}_i]$, we define for $i = 1, \dots, n$, $\xi_i =_{\text{df}} \xi_{i-1} \wedge (\psi_i[\mathbf{x}_i \mapsto \mathbf{v}_i])$ with $\xi_0 = \top$, and

$$N'_D(\xi_i) =_{\text{df}} N'_D(\xi_{i-1}) \boxtimes N'(\psi_i)[\mathbf{x}_i \mapsto \mathbf{v}_i]. \quad (5)$$

We set $N'(D) =_{\text{df}} \text{Res}(N_D(\phi_D)) = \text{Res}(N_D(\xi_n))$ to be the result of our translation into P/T nets.

4 A small Case Study

To validate our approach we performed a small case study with the unavoidable ACME travel workflow. We analyzed the DecSerFlow variant of this standard example [32]. Fig. 14 shows the textual representation of the DSF workflow $D_{\text{travel}} = (\text{Tasks}_{\text{tr}}, \phi_{\text{tr}})$:

We implemented a tool that reads a textual specification of a DSF workflow as in Fig. 14, looks up the corresponding Petri net pattern in a library, and depending on the chosen Petri net type, generates an IR net based on (3) or a P/T net based on (5). The patterns are stored in PNML format [4], output of the tool is PNML as well. The result of the translating D_{travel} to IR nets (laid out manually and structurally reduced by removing redundant places like $\text{pre}_{\text{receive}}$) is depicted in Fig. 15.

$$\begin{aligned}
Task_{tr} &= \{receive, hotel, bookedHotel, failedHotel, airline, bookedAirline, failedAirline, \\
&\quad creditCard, notifyBooked, compensation, notifyFailure\} \\
\phi_{tr} &= ex_1^1(receive) \wedge precedence(receive, hotel) \wedge precedence(receive, airline) \\
&\quad \wedge response(receive, hotel) \wedge response(receive, airline) \\
&\quad \wedge precedence(hotel, bookedHotel) \wedge precedence(hotel, failedHotel) \\
&\quad \wedge response_2(hotel, \{bookedHotel, failedHotel\}) \\
&\quad \wedge precedence(airline, bookedAirline) \wedge precedence(airline, failedAirline) \\
&\quad \wedge response_2(airline, \{bookedAirline, failedAirline\}) \\
&\quad \wedge precedence(bookedHotel, creditCard) \wedge precedence(bookedAirline, creditCard) \\
&\quad \wedge ex_0^1(creditCard) \wedge notResponse(creditCard, hotel) \wedge notResponse(creditCard, airline) \\
&\quad \wedge precedence(creditCard, notifyBooked) \wedge response(creditCard, notifyBooked) \\
&\quad \wedge ex_0^1(notifyBooked) \\
&\quad \wedge precedence_2(\{failedHotel, failedAirline\}, compensation) \wedge ex_0^1(compensation) \\
&\quad \wedge notResponse(compensation, hotel) \wedge notResponse(compensation, airline) \\
&\quad \wedge precedence(compensation, notifyFailure) \wedge ex_0^1(notifyFailure) \\
&\quad \wedge notCoExistence(creditCard, notifyFailure) \wedge mutualSubst(creditCard, notifyFailure)
\end{aligned}$$

Fig. 14. DecSerFlow specification of the ACME travel workflow

Analysis. We used the P/T net $N'(D_{travel})$, containing 74 places and 35 transitions and therefore not being shown here, for an analysis with Petri net based tools. Verifying that the generated net is weakly terminating (reaching a marking with a token in finish is always possible) [19] with the CTL model checker LoLA [28] revealed a life-lock in $N'(D_{travel})$ that is caused by the firing sequence

$$\langle receive, hotel, failedHotel, compensation, notifyFailure \rangle.$$

Because our IR and P/T net patterns are branching bisimilar, and therefore equivalent wrt CTL*-X properties [20], the same firing-sequence also causes a life-lock in $N(D_{travel})$: In this life-lock `bookedHotel` and `failedHotel` may fire infinitely often. Transition `end` has no concession because `pendingairline` is still marked while `disallowsairline` contains a token which blocks `airline`.

Our mechanism is equivalent to the Büchi automata translation in the sense that both models include the non-accepting executions of the LTL specification. We have such a case here: The life-lock is a non-accepting firing sequence of $N'(D_{travel})$. However, the safety properties in ϕ_{tr} cannot prevent a non-accepting execution due to violated liveness properties if one bases the choice of the next task on the current state only. Therefore D_{travel} specifies a not weakly terminating system.

Exploiting our compositional approach, we performed backwards analysis on the involved places and transitions and identified the involved constraints. The life-lock is caused by `response(receive, airline)` and `notResponse(compensation, airline)`: The problem with D_{travel} is, that for `compensation` to occur only one of `airline` and `hotel` need to be executed beforehand. The occurrence of the former task blocks the latter two tasks. But the occurrence of `receive` must be followed by `airline` and

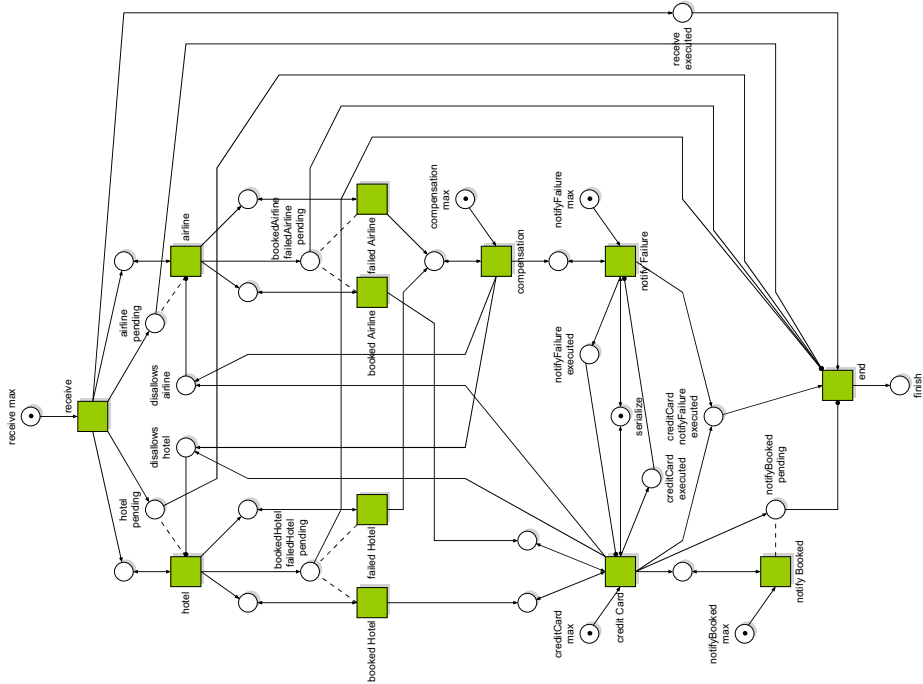


Fig. 15. Translated ACME travel workflow

hotel. Thus in any accepting execution, of D_{travel} , occurrence of **compensation** must be preceded by **hotel** and **airline**.

One may argue that this logical property can be deduced by careful inspection of the LTL constraints or application of a theorem prover. But one needs to know the property to be deduced beforehand, to see if it can be deduced. In contrast, for identifying unknown problems of a system, as exercised here, standard techniques for operational models are applicable [10]. In this sense, identifying an unsafe underspecification of the DSF workflow was greatly simplified through its equivalent operational Petri net model.

We can think of two solutions to complete the specification. We could explicitly require that **compensation** is preceded by **hotel** and **airline**. We prefer to allow **compensation** as an alternative response to **receive**: We refine D_{travel} and use $\text{response}_2(\text{receive}, \{\text{airline}, \text{compensation}\})$ instead of $\text{response}(\text{receive}, \text{airline})$ (and $\text{response}_2(\text{receive}, \{\text{hotel}, \text{compensation}\})$ instead of $\text{response}(\text{receive}, \text{hotel})$ for symmetry reasons) in the specification, let D_{travel}^2 denote the refined workflow.

We can see from $N(\text{response}_2(A, \{B_1, B_2\}))$ in Fig. 9 that this results in extending $N(D_{\text{travel}})$ of Fig. 15 with reset arcs (**compensation**, **pending_{airline}**) and (**compensation**, **pending_{hotel}**). Model-checking $N'(D_{\text{travel}}^2)$ verifies the soundness of D_{travel}^2 : the refined workflow can always terminate.

5 Conclusion

We presented a compositional mechanism to synthesize a Petri net from a LTL specification for flexible workflows. The synthesized Petri net contains the intuition of the LTL specification. Further, it equivalently implements the specification in the sense that each path satisfying the specification corresponds to a firing sequence of the Petri net that leads to an accepting marking and vice versa. The synthesizing translation is compositional: each constraint is translated to a Petri net pattern; conjoining constraints translates to merging the corresponding nets. A key aspect for the equivalence of the synthesized Petri net is our ‘canvas’ that implements the empty specification.

The mechanism can synthesize specifications being conjunctions of known constraints only as the mapping to a Petri net is based on the intention of the constraint and not on the structure of its LTL formula. We have shown that through the compositionality of our mechanism, analysis results performed at the Petri net level can be translated back to the LTL formula. Further, our approach preserves the logical structure of the workflow as the resulting Petri net reflects an understandable process model. Therefore applying Petri net specific analysis techniques like invariants, siphons and traps [29] can provide information about the workflow itself. This property makes our approach superior in the workflow domain compared to standard translations like the the generation of a Büchi automaton.

Related Works. The synthesis of Petri nets from specifications involving logics is for instance studied in the field of ‘controller synthesis’. Yet the setting differs as an existing system shall be controlled to meet the specification which does not correspond to our setting. We are not aware of any general translation mechanisms from LTL formulas (or other declarative formalisms) to Petri nets. More importantly, we think that even if such a mechanism existed, the resulting Petri net would present as much intuition of the system as a synthesized Büchi automaton.

In the context of workflows and business processes our work relates to [18] where the behavior of a workflow process is restricted by constraints. Formally this is done by merging a Petri net workflow model with Petri nets implementing the constraints. Our approach is more general as we don’t require an existing Petri net; further the constraints considered in either work are different. Another approach is a logic to specify contracts over sets of web services [8] where we differ in applying established formalisms like LTL and Petri nets.

Future Work. Open problems in our approach are an automated translation of analysis results in the operational model back to the declarative model. Furthermore, a less manual approach in deriving the Petri net patterns from LTL constraints would be desirable to address the extensibility of DecSerFlow. Besides real-world case studies from application domains like disaster management, it is certainly necessary to consider open systems that communicate with their environment. A candidate logic would be alternating-time temporal logic (ATL) [3] that allows the synthesis of a finitely representable operational model [14].

Acknowledgements. I would like to thank Niels Lohmann for providing me the libraries and algorithms that allowed me to implement the tool for the case study.

References

- [1] W.M.P. van der Aalst. The Application of Petri Nets to Workflow Management. *The Journal of Circuits, Systems and Computers*, 8(1):21–66, 1998.
- [2] Alessandra Agostini and Giorgio De Michelis. Improving flexibility of workflow management systems. In *Business Process Management, Models, Techniques, and Empirical Studies*, pages 218–234, London, UK, 2000. Springer-Verlag.
- [3] Rajeev Alur, Thomas A. Henzinger, and Orna Kupferman. Alternating-time temporal logic. In *COMPOS'97: Revised Lectures from the International Symposium on Compositionality: The Significant Difference*, pages 23–60, London, UK, 1998. Springer-Verlag.
- [4] Jonathan Billington, Søren Christensen, Kees M. van Hee, Ekkart Kindler, Olaf Kummer, Laure Petrucci, Reinier Post, Christian Stehno, and Michael Weber. The petri net markup language: Concepts, technology, and tools. In *ICATPN*, pages 483–505, 2003.
- [5] F. Casati, P. Grefen, B. Pernici, G. Pozzi, and G. Sanchez. Wide workflow model and architecture (ctit 96-19). Technical report, University of Twente, 1996.
- [6] Fabio Casati, Stefano Ceri, Barbara Pernici, and Giuseppe Pozzi. Workflow evolution. In *International Conference on Conceptual Modeling / the Entity Relationship Approach*, pages 438–455, 1996.
- [7] Constantin Courcoubetis, Moshe Y. Vardi, Pierre Wolper, and Mihalis Yannakakis. Memory-efficient algorithms for the verification of temporal properties. *Formal Methods in System Design*, 1(2/3):275–288, 1992.
- [8] Hasan Davulcu, Michael Kifer, and I.V. Ramakrishnan. CTR-S: A logic for specifying contracts in semantic web services. In *WWW2004*, pages 144+, 2004.
- [9] C. Dufourd, A. Finkel, and Ph. Schnoebelen. Reset nets between decidability and undecidability. In Kim Guldstrand Larsen, Sven Skyum, and Glynn Winskel, editors, *ICALP'98*, volume 1443 of *LNCS*, pages 103–115, 1998.
- [10] Jr. Edmund M. Clarke, Orna Grumberg, and Doron A. Peled. *Model Checking*. MIT Press, Cambridge, MA, USA, 1999.
- [11] Clarence Ellis, Karim Keddara, and Grzegorz Rozenberg. Dynamic change within workflow systems. In *COCS '95: Proceedings of conference on Organizational computing systems*, pages 10–21, New York, NY, USA, 1995. ACM Press.
- [12] Dov Gabbay, Amir Pnueli, Saharon Shelah, and Jonathan Stavi. On the temporal analysis of fairness. In *POPL '80: Proceedings of the 7th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 163–173, New York, NY, USA, 1980. ACM Press.
- [13] Rob Gerth, Doron Peled, Moshe Y. Vardi, and Pierre Wolper. Simple on-the-fly automatic verification of linear temporal logic. In *Protocol Specification Testing and Verification*, pages 3–18, Warsaw, Poland, 1995. Chapman & Hall.
- [14] Valentin Goranko and Govert van Drimmelen. Complete axiomatization and decidability of alternating-time temporal logic. *Theor. Comput. Sci.*, 353(1):93–117, 2006.

- [15] H. Ben-Abdallah and S. Leue. Syntactic detection of process divergence and non-local choice in message sequence charts. In E. Brinksma, editor, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 259–274, Enschede, The Netherlands, 1997. Springer Verlag, LNCS 1217.
- [16] Petra Heintl, Stefan Horn, Stefan Jablonski, Jens Neeb, Katrin Stein, and Michael Teschke. A comprehensive approach to flexibility in workflow management systems. In *WACC '99*, pages 79–88, New York, NY, USA, 1999. ACM Press.
- [17] Niels Lohmann, Peter Massuthe, Christian Stahl, and Daniela Weinberg. Analyzing Interacting BPEL Processes. In *BPM 2006, Proceedings*, volume 4102 of *LNCS*, pages 17–32. Springer-Verlag, September 2006.
- [18] Niels Lohmann, Peter Massuthe, and Karsten Wolf. Behavioral constraints for services. Informatik-Berichte 214, Humboldt-Universität zu Berlin, May 2007.
- [19] Peter Massuthe and Karsten Wolf. Operating Guidelines for Services. *Petri Net Newsletter*, 70:9–14, April 2006.
- [20] Rocco De Nicola and Frits Vaandrager. Three logics for branching bisimulation. *J. ACM*, 42(2):458–487, 1995.
- [21] Amir Pnueli. The temporal logic of programs. In *FOCS*, pages 46–57, 1977.
- [22] Manfred Reichert and Peter Dadam. Adept_{flex}-supporting dynamic changes of workflows without losing control. *J. Intell. Inf. Syst.*, 10(2):93–129, 1998.
- [23] Wolfgang Reisig. *A Primer in Petri Net Design*. Springer Compass International, 1992.
- [24] Stefanie Rinderle, Manfred Reichert, and Peter Dadam. Evaluation of correctness criteria for dynamic workflow changes. In *Business Process Management*, pages 41–57, 2003.
- [25] Stefanie Rinderle, Manfred Reichert, and Peter Dadam. Correctness criteria for dynamic changes in workflow systems: a survey. *Data Knowl. Eng.*, 50(1):9–34, 2004.
- [26] Shazia W. Sadiq, Wasim Sadiq, and Maria E. Orlowska. Pockets of flexibility in workflow specification. In *ER*, pages 513–526, 2001.
- [27] Shazia Wasim Sadiq, Maria E. Orlowska, and Wasim Sadiq. Specification and validation of process constraints for flexible workflows. *Inf. Syst.*, 30(5):349–378, 2005.
- [28] Karsten Schmidt. Lola: A low level analyser. In *ICATPN*, pages 465–474, 2000.
- [29] Peter H. Starke. *Analyse von Petri-Netz-Modellen*. Stuttgart, Germany: Teubner, 1990. NewsletterInfo: 38.
- [30] W. M. P. van der Aalst and T. Basten. Inheritance of workflows: an approach to tackling problems related to change. *Theor. Comput. Sci.*, 270(1-2):125–203, 2001.
- [31] W. M. P. van der Aalst and S. Jablonski. Dealing with workflow change: identification of issues and solutions. *CSSE*, 15(5):267–276, September 2000.
- [32] Wil M. P. van der Aalst and Maja Pesic. Decserflow: Towards a truly declarative service flow language. In *WS-FM*, pages 1–23, 2006.
- [33] Wil M. P. van der Aalst and Kees M. van Hee. *Workflow Management: Models, Methods, and Systems*. MIT Press, 2002.
- [34] Gottfried Vossen and Mathias Weske. The wasa2 object-oriented workflow management system. In *SIGMOD '99: Proceedings of the 1999 ACM SIGMOD international conference on Management of data*, pages 587–589, New York, NY, USA, 1999. ACM Press.
- [35] WfMC. Terminology and glossary, 3rd edition. Technical report, Workflow Management Coalition, Winchester, 1999.