A full R/I-net construct lexicon for declare constraints

De Smedt J, vanden Broucke S, De Weerdt J, Vanthienen J.

KBI_1506

A Full R/I-net Construct Lexicon for Declare Constraints

Johannes De Smedt, Seppe K.L.M. vanden Broucke, Jochen De Weerdt, and Jan Vanthienen

KU Leuven Faculty of Economics and Business
Department of Decision Sciences and Information Management
Naamsestraat 69
B-3000 Leuven, Belgium
`firstname.lastname@kuleuven.be`

Technical report. Recently, declarative process languages are gaining ground as the procedural way of capturing activities in a fixed workflow is deemed to be inflexible. Declare, one of the prime languages of the declarative process modeling paradigm, is composed of logic-based activity constraints which are event-driven. In this paper, a template-based conversion is proposed of every Declare constraint into a single Petri net fragment with weighted, reset and inhibitor arcs, i.e. a weighted R/I-net. As such, a formalization of the execution semantics of Declare is obtained, similar to linear temporal logic or regular expressions, but now expressed in the form of Petri nets. Equivalence of Declare constraints and the respective Petri net templates are analyzed at the theoretical level and by means of a simulation experiment.
Keywords: Declare, conversion, weighted R/I-nets, formalization, execution semantics, reset arcs, inhibitor arcs

1 Introduction

Petri nets are a widely used language to express concurrent systems [27]. They are highly expressive and have a clear execution semantics due to sound mathematical underpinnings, which results in a wide availability of robust analysis techniques. Therefore, in the context of business processes, Petri nets are extensively used for model design and verification [2], so that even more user-centric languages such as Business Process Modeling and Notation (BPMN) [11] and Event-driven Process Chains (EPCs) [1] are frequently converted to some form of Petri net for checking model properties such as soundness [2, 17].

The need for flexibility in business processes, however, has led to an increase in popularity of declarative models. Approaches found in literature include Declare [28], pockets of flexibility, [32], workflows mixed with data rules [22], and worklets [7]. A comprehensive overview can be found in [20]. Most notably, Declare has become the de facto standard for modeling flexible workflows by using an event-driven, constraint-based approach. Mixed forms of both paradigms are also gaining ground. YAWL [4], a language which extends Petri nets with more flexible constructs such as resetting capabilities, can include Declare constraints in subworkflows [3], a more advanced version of ad-hoc subprocesses in BPMN. A real mix of state-spaces of Petri nets and Declare is proposed and implemented in [37].

This paper presents a template-based technique that converts every Declare constraint into a weighted R/I-net, i.e. a Petri net with weighted, reset and inhibitor arcs. This conversion results in a formalization of Declare, similar to Linear Temporal Logic (LTL) [13] or regular expressions [38], but one expressed with Petri nets. As such, we obtain a Petri net template lexicon that expresses the same behavior as the collection of Declare constraints originally described in the seminal work of Pesic [28].

The conversion of Declare models to Petri nets has been touched upon in [18], proposing R/I-net constructs for a subset of DecSerFlow constraints [5], the predecessor of Declare. Also, the synthesis to a Petri net without R/I-constructs is proposed in an example. Our approach differs by putting forward a full lexicon of conversions and a thorough analysis of their applications. A full synthesis approach is proposed in [30], where Declare constraints are redefined as regular expressions, converted to finite state machines and finally synthesized into Petri nets with the theory of regions [12]. While the conversion strategy in [30] is interesting, our approach yields several benefits. First, while the automata of the regular expressions need to be multiplied for Declare models consisting of multiple constraints before synthesis, our technique works with separable Petri net templates (thus regions), which makes converting a full Declare model a simple addition of templates. Second, synthesis will provoke duplication of tasks (i.e. transitions with the same label) which results in (i) nets that are arguably even harder to read than automata, (ii) increased complexity for analysis tasks such as conformance checking, and (iii) impediments for straightforward plugging of Petri net fragments that could be obtained by synthesizing every constraint separately with the technique presented in [30]. Third, the use of inhibitor arcs to withhold an ending activity from firing when a constraint is violated as done in this work makes the specification and keeping track of violation easier. In addition, every violation can be traced back to a certain place and activity.

The remainder of this paper is structured as follows. First, the preliminaries section contains a description of the semantics of both Declare (in LTL and regular expressions) and weighted R/I-nets. Section 3 describes in full the conversion lexicon of all Declare constraints into Petri net constraints. Section 4 contains an analysis of equivalence and empirical validation, followed by the conclusion.

2 Preliminaries

This section provides a brief overview of concepts and definitions used throughout the paper.

2.1 Declare

Declare [28] is a framework that was originally proposed by [29]. The framework consists of a declarative process language called ConDec [5], which itself is based on Linear Temporal Logic, and a model checking framework for rule verification [36]. ConDec lists a number of constraints, which are usually categorized into seven groups, i.e. the *Unary*, *Binary existence*, *Simple ordered*, *Alternating ordered*, *Chain ordered*, *Negative*, and *Choice* constraints. A full overview of all constraints can be found in Table 1.

In order to execute Declare models, i.e., a set of declarative constraints, the constraints are converted to Büchi automata [33]. This conversion is valid, as long as the LTL constraints are insensitive to infiniteness. This has been proven in [14] for all Declare constraints but one, *Not chain succession*, which originally has been formulated incorrectly. In this work, we interpret the constraint here as the correct, finite variant, i.e. as $\Box(A \Rightarrow \neg(\bigcirc B))$.

Next, by taking the product of all separate automata (one for each constraint), a full executable model is obtained, which can then be applied to detect satisfying, temporal, and permanently violated states when replaying words over them [25].

In recent work, a shift towards expressing Declare constraints by means of regular expressions (as opposed to LTL formula) is witnessed in works such as [16, 38]. Both deem LTL unfit to express finite traces and hence redefine Declare in finite state machines (although a formal proof of similarity between the behavior expressed in the FSMs and Büchi automata is as of yet not provided).

2.2 Weighted R/I-nets

In this paper, we propose a formalization to express Declare constraints in the form of weighted Petri nets with reset and inhibitor arcs. Petri nets [27] are a mathematical modeling language to describe distributed, concurrent systems. A weighted Petri net with reset and inhibitor arcs is a directed graph, expressed as a tuple, $PN = (P, T, F, R, I, W)$, with P a finite set of places (visually represented as circles), T a finite set of transitions (visually represented as boxes) with $P \cap T = \emptyset$, and $F \subseteq (P \times T) \cup (T \times P)$ the set of normal arcs (shown as arcs with a single arrow). Let $W : F \rightarrow \mathbb{N}$ determine a weighting function which associates a weight to each arc. Let $R : T \rightarrow \mathbb{P}(P)$ define the reset places (with $\mathbb{P}(P)$ the powerset of P) and $I : T \rightarrow \mathbb{P}(P)$ the inhibitor places for each transition, which also implicitly define the reset arcs (shown as an arc ending with double arrows) and inhibitor arcs (shown as an arc ending with a circle) respectively. The set of input nodes of a node $x \in P \cup T$ is denoted as $\bullet x = \{(y \in P \cup T | (y, x) \in F) \vee (x \in T \wedge y \in R(x) \cup I(x))\}$, and the output nodes similarly as $x\bullet$.

The state of a Petri net is called marking $M \in P \rightarrow \mathbb{N}$, indicating the number of tokens contained in each place. A transition t is said to be enabled, denoted as $M[t\rangle$, iff $M(p) > 0, \forall p \in \bullet t : [(p, t) \in F \vee p \in R(t)] \wedge M(p) =$

Table 1. An overview of Declare constraint templates with their corresponding LTL formula, regular expression, and R/I-net constructs.

Template	LTL Formula [29]	Regular Expression [38]	R/I-Net Constructs	Description
Existence(A,n)	$\Diamond(A \wedge \bigcirc(existence(n-1, A)))$	.*(A.*){n}	$F = \{(A, p_1), (p_1, t_{sink})\}, W(p_1, t_{sink}) = n$	Activity A happens at least n times.
Absence(A,n)	$\neg existence(n, A)$	[^A]*(A?[^A]*){n-1}	$F = \{(t_{source}, p_1), (p_1, t_A)\}, W(t_{source}, p_1) = n-1$	Activity A happens at most n times.
Exactly(A,n)	$existence(n, A) \wedge absence(n+1, A)$	[^A]*(A[^A]*){n}	$F = \{(t_{source}, p_1), (p_1, t_A)\}, W(t_{source}, p_1) = n, I(t_{sink}) = p_1$	Activity A happens exactly n times.
Init(A)	A	(A.*)?	$F = \{(t_{source}, p_1)\}, \forall t \in T \setminus t_A, I(t) = p_1$	Each instance has to start with activity A.
Last(A)	$\Box(A \implies \neg X \neg A)$	.*A	$F = \{(t_A, p_1), (p_1, t_{sink})\}, \forall t \in T \setminus t_A, R(t) = p_1$	Each instance has to end with activity A.
Responded existence(A,B)	$\Diamond A \implies \Diamond B$	[^A]*((A.*B.*) |(B.*A.*))?	$F = \{(t_A, p_1), (t_B, p_2), (p_2, t_\lambda), (t_\lambda, p_3)\}, I(t_A) = p_3, I(t_B) = p_3, I(t_{sink}) = p_1, R(t_\lambda) = \{p_1, p_2\}$	If A happens at least once then B has to happen or happened before A.
Co-existence(A,B)	$\Diamond A \impliedby \Diamond B$	[^AB]*((A.*B.*) |(B.*A.*))?	$F = \{(t_A, p_1), (t_A, p_4), (t_B, p_2), (t_B, p_4), (p_1, t_\lambda), (p_2, t_\lambda), (t_\lambda, p_3)\}, I(t_A) = p_3, I(t_B) = p_3, I(t_{sink}) = p_4, R(t_\lambda) = \{p_1, p_2, p_4\}$	If A happens then B has to happen or happened after after A, and vice versa.
Response(A,B)	$\Box(A \implies \Diamond B)$	[^A]*(A.*B)*[^A]*	$F = \{(t_A, p_1)\}, I(t_{sink}) = p_1, R(t_B) = p_1$	Whenever activity A happens, activity B has to happen eventually afterward.
Precedence(A,B)	$(\neg B\, U A) \vee \Box(\neg B)$	[^B]*(A.*B)*[^B]*	$F = \{(t_A, p_1), (t_B, p_1), (p_1, t_B)\}$	Whenever activity B happens, activity A has to have happened before it.
Succession(A,B)	$response(A, B) \wedge precedence(A, B)$	[^AB]*(A.*B)*[^AB]*	$F = \{(t_A, p_1), (t_B, p_1), (p_1, t_B), (t_A, p_2)\}, I(t_{sink}) = p_2, R(t_B) = p_2$	Both Response(A,B) and Precedence(A,B) hold.
Alternate response(A,B)	$\Box(A \implies \bigcirc(\neg A\, U\, B))$	[^A]*(A[^A]*B[^A]*)*	$F = \{(t_{source}, p_1), (p_1, t_A), (t_A, p_2), (t_B, p_1)\}, I(t_{sink}) = p_2, R(t_A) = p_1, R(t_B) = p_2$	After each activity A, at least one activity B is executed. A following activity A can be executed again only after the first occurrence of activity B.
Alternate precedence(A,B)	$precedence(A, B) \wedge \Box(B \implies \bigcirc(precedence(A, B))$	[^B]*(A[^B]*B[^B]*)*	$F = \{(t_A, p_1), (p_1, t_B)\}, R(t_B) = p_1$	Before each activity B, at least one activity A is executed. A following activity B can be executed again only after the first next occurrence of activity A.
Alternate succession(A,B)	$altresponse(A, B) \wedge precedence(A, B)$	[^AB]*(A[^AB]*B[^AB]*)*	$F = \{(t_{source}, p_1), (p_1, t_A), (t_A, p_2), (p_2, t_B), (t_B, p_1)\}, I(t_{sink}) = p_2$	Both alternative response(A,B) and alternate precedence(A,B) hold.
Chain response(A,B)	$\Box(A \implies \bigcirc B)$	[^A]*(AB[^A]*)*	$F = \{(t_A, p_1)\}, I(t_A) = p_1, I(t_C) = p_1, R(t_B) = p_1$	Every time activity A happens, it must be directly followed by activity B (activity B can also follow other activities).
Chain precedence(A,B)	$\Box(\bigcirc B \implies A)$	[^B]*(AB[^B]*)*	$F = \{(t_{source}, p_1), (t_B, p_1), (t_C, p_1)\}, I(t_B) = p_1, R(t_A) = p_1$	Every time activity B happens, it must be directly preceded by activity A (activity A can also precede other activities).
Chain succession(A,B)	$\Box(A \iff \bigcirc B)$	[^AB]*(AB[^AB]*)*	$F = \{(t_{source}, p_2), (t_A, p_1), (t_B, p_2)\}, I(t_A) = p_1, I(t_B) = p_2, I(t_C) = p_1, R(t_A) = p_2, R(t_B) = p_1$	Activities A and B can only happen directly following each other.
Not co-existence(A,B)	$\neg(\Diamond A \wedge \Diamond B)$	[^AB]*((A[^B]*) |(B[^A]*))?	$F = \{(t_A, p_1), (t_B, p_2)\}, I(t_A) = p_2, I(t_B) = p_1$	Either activity A or B can happen, but not both.
Not succession(A,B)	$\Box(A \implies \neg(\Diamond B))$	[^A]*(A[^B]*)*	$F = \{(t_A, p_1)\}, I(t_B) = p_1$	Activity A cannot be followed by activity B, and activity B cannot be preceded by activity A.
Not chain succession(A,B)	$\Box(A \implies \neg(\bigcirc B))$	[^A]*(A+[^AB][^A]*)*A*	$F = \{(t_A, p_1)\}, I(t_B) = p_1, R(t_C) = p_1$	Activities A and B can never directly follow each other.
Choice(A,B)	$\Diamond A \vee \Diamond B$	.*[AB].*	$F = \{(t_{source}, p_1)\}, I(t_{sink}) = p_1, R(t_A) = p_1, R(t_B) = p_1$	Activity A or activity B has to happen at least once, possibly both.
Exclusive choice(A,B)	$(\Diamond A \vee \Diamond B) \wedge \neg(\Diamond A \wedge \Diamond B)$	([^B]*A[^B]*) |.*[AB].*([^A]*B[^A]*)	$F = \{(t_{source}, p_2), (t_A, p_1), (t_B, p_3)\}, I(t_A) = p_3, I(t_B) = p_1, I(t_{sink}) = p_2, R(t_A) = p_2, R(t_B) = p_2$	Activity A or activity B has to happen at least once, but not both.

$0, \forall p \in I(t)$. Firing an enabled transition results in a new marking M' so that $M'(p) = M(p) - (M(p)$ iff $p \in R(t), W(p,t)$ iff $(p,t) \in F, 0$ otherwise$) + (W(t,p)$ iff $(t,p) \in F, 0$ otherwise$)$. That is, tokens are removed from input places according to arc weights. Places which act as reset places for a fired transition are emptied completely. Next, the token count of output places is incremented according to arc weights to obtain the new marking. We refer to [27] for more details.

3 Conversion of Declare Constraints to Weighed R/I-nets

This section describes the conversion of each Declare constraint into their dedicated weighted R/I-nets. It also provides a section on equivalence analysis along with an empirical evaluation thereof.

3.1 Conversion templates

Since the purpose of the conversions is to capture constraints which are usually expressed in (Büchi) automata that yield (ω-)regular languages, we seek to construct Petri nets which produce regular languages. This is the case when the net is in clean standard form [21]. Hence, the following guiding principles are followed.

For every letter in the Declare template/model alphabet, we define one in the Petri net alphabet $\Sigma_{PN} = \Sigma_{Dec} \cup \{\lambda_{Invisible}, \lambda_{Start}, \lambda_{End}\}$, with labeling function $\delta : T \rightarrow \Sigma_{PN}$. As such, PN has a non-lamda free language L^{λ}. We use invisible transitions (labeled $\lambda_{Invisible}$) for two constraints and will refer to λ_{Start} and λ_{End} as *Start* and *End* respectively. We refer to their corresponding nodes as t_{source} and t_{sink} with $\delta(t_{source}) = \lambda_{Start}$ and $\delta(t_{sink}) = \lambda_{End}$. t_{source} is used to start the net by filling helper places used for the constraints, which need tokens to inhibit constraints which are (temporarily) violated by default. For example, an *Existence(A,n)* constraint puts the model in a temporarily violated state when initialized, only to reach an accepting state after activity A has fired at least n times. t_{sink} empties the net completely and functions as an indicator for the state of the net as well. If t_{sink} is enabled, this means the net is in an accepting state, otherwise, there are places tied to constraints inhibiting it from firing. The end transition empties all places upon firing by using reset arcs, leaving the net in its final marking, comprising of only one token in the final place p_{sink}. Hence PN has an *L-type* ending. According to [21], as all template nets are in clean standard form, they yield a regular language. Indeed, unless the net is in a violated state (marking) which cannot reach an accepting state anymore (permanent violation), there exists a firing sequence reachable from that marking which ends with the sink transition. t_{sink} empties the net and is the sole terminal on the right hand side of that marking. Also, since none of the binary constraints contain any cardinality-based execution patterns, none of the Declare Petri nets need context-free grammar constructs such as $a^n b^n$. The R/I-net constructs for each template can be found in Table 1. In the construct templates, it is assumed that $T = \{t_{sink}, t_A(, t_B, t_C), t_{sink}\}, t_C = T \setminus \{t_A, t_B, t_{source}, t_{sink}\}$, $P = \{p_{source}, p_{sink}, p_1(, p_2, p_3, p_4)\}$, and $F = \{(p_{source}, t_{source}), (t_{sink}, p_{sink})\}$.

Note that the *Init* and *Last* constraints cannot be substituted by t_{source} and t_{sink}, unless the activities these constraints refer to also have to execute exactly once. Otherwise, they would recreate the initial marking over and over again (*Init*, as t_{source} initializes the net), or be inhibited from firing multiple times (*Last*, as t_{sink} empties the net). As such, *Init(A)* is instead modeled as a fragment containing a single place which is filled by t_{source} and inhibits every transition but t_A. The *Last(A)* constraint is enforced by a single place which gets filled by t_A and is reset by all the other transitions but t_A.

Many helper places are introduced which are used to enforce and indicate the state of the constraints in a one-to-one fashion. This means that one can trace back every constraint in the net to a certain Declare template. This is not possible for example, in the approach of [30], but yields benefits for, e.g., model and conformance checking, where in an execution the violation can be traced back to a certain constraint and the activities involved.

Figure 1 provides an example of converting a full Declare model in standard notation to its converted Petri net counterpart after applying our formalization. For this purpose, we included a Declare model with three unary and three binary constraints. *Deposit money* is executed first, and at least once. *Request credit card* can only occur exactly afterwards *Deposit money* and exactly once. *Deposit money* precedes *Withdraw money.* Every new occurrence of *Print statement* must be preceded by at least one new occurrence of *Withdraw money.* Note that opportunities exist for simplifying the resulting Petri net model, which was not done here for the sake of understandability of the separate converted constraints; in the following paragraphs, we discuss the conversion of each of the Declare constraint types to their corresponding Petri net templates and how to create a model out of a full Declare model (a set of constraints). Note that we will refer to the activities involved in binary constraints as antecedent (A) and consequent (B).

Unary Constraints. Unary constraints (shown in Figure 2) focus mainly on the enforcement of cardinalities of the activity involved. In these scenarios, the source and sink transitions prove useful already. *Existence(A,n)* requires an activity A to be executed at least n times. Therefore, a helper place p_1 is installed, that can only fire after n tokens are collected in p_1, by introducing an arc with weight n connected to t_{sink}.

Absence(A,n) translates into putting $n-1$ tokens in p_1, which will prevent the activity from firing n or more times. The *Exactly(A,n)* constraint is similar, but p_1 inhibits t_{sink} from firing until the activity has happened at least n times.

Binary Existence Constraints. These constraints (shown in Figure 3) are the hardest to capture in a Petri net template, as they require some sort of memory to keep track of the fact whether the antecedent and consequent have occurred before. For *Responded existence*, the net cannot end before a first occurrence

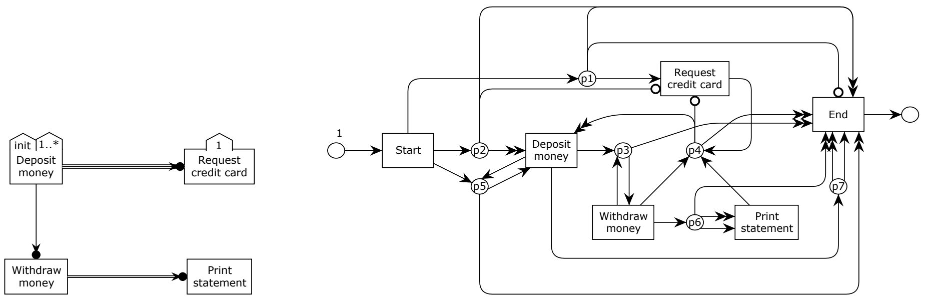

Fig. 1. Simple Declare model containing 3 unary and 3 binary constraints to the left and the converted net to the right. The net contains 7 places (besides p_{source} and p_{sink}) of which 6 are used for the constraints and one, p_5, which adds a self-loop for *Deposit Money.* p_1 is used for *Exactly(Request credit card,1)*, p_2 for *Init(Deposit money)*, p_3 for *Precedence(Deposit money, Withdraw money)*, p_4 for *Chain precedence(Deposit money, Request credit card)*, p_6 for *Alternate precedence(Withdraw money, Print statement)*, and p_7 for *Existence(Deposit money,1).* Many reset arcs are connected from these places to t_{sink} to empty the net upon firing this transition.

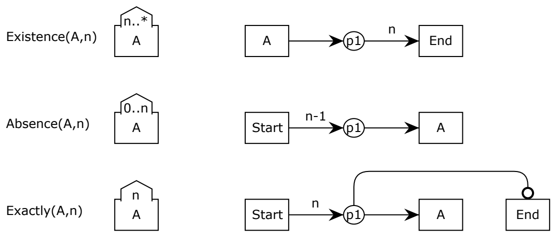

Fig. 2. The mapping of unary constraints *Existence(A,n)*, *Absence(A,n)*, and *Exactly(A,n).* Note that the t_{source} and t_{sink} transitions (labeled *Start* and *End*) are only shown when necessary in this and following figures.

of the consequent when the antecedent fires. The net thus has to keep track whether the consequent has fired already, in case a termination is sought for after the firing of the antecedent. For this purpose, an invisible activity is used which, when fired, leaves no transition enabled but the last one, acting as a placeholder for t_{sink}. The same principle can be applied for *Co-existence*, which is the two-way version of *Responded Existence.*

Simple Ordered Constraints. *Response* requires a helper place that inhibits t_{sink} as long as the consequent of the constraint is not fired after an occurrence of the antecedent. *Precedence* also needs an additional input place for the consequent, which serves to enabled it after the firing of the antecedent. Afterwards, the consequent keeps itself enabled indefinitely. *Succession* is, similar to the LTL formula, the combination of both constraints. The constraints are shown in Figure 4.

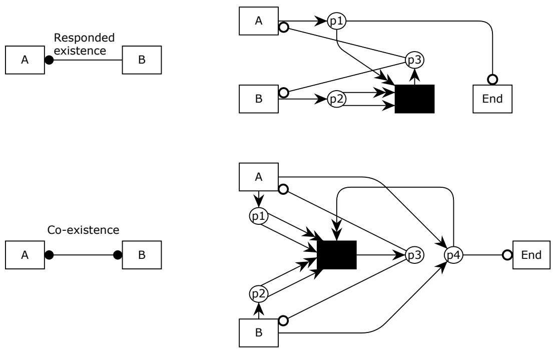

Fig. 3. The mapping of the binary existence constraints *Responded*, and *Co-existence.*

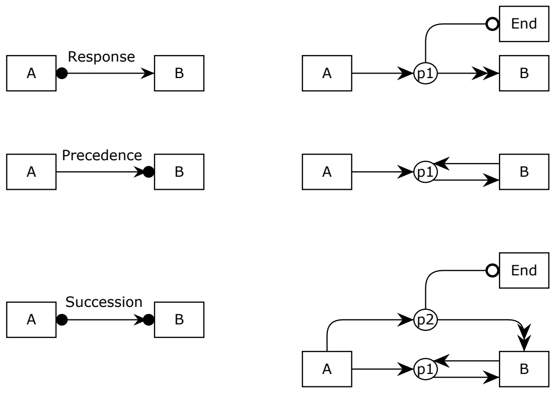

Fig. 4. The mapping of the simple ordered constraints *Response, Precedence*, and *Succession.*

Alternating Ordered Constraints. Alternating constraints (shown in Figure 5) are an LTL-based way to express loops. Modeling *Alternate response* is somewhat tedious. First of all, two helper places need to be introduced, p_1 and p_2. p_1 is an input place of t_{source}, resulting in an initial marking of 1. Next, it serves both as an input for the antecedent, and also a reset place. The consequent can fire any time, but whenever the antecedent is fired, t_{sink} is inhibited until the occurrence of the consequent, which frees p_2 with a reset operation. Also, the consequent delivers a new token to the input place of the antecedent. *Alternate precedence* is much more straightforward. Whenever the consequent is fired, it resets its input place, which models the fact that it can only occur after any new occurrence of the antecedent. Again, *Alternate succession* is a combination of the two, but can be reduced to a smaller mapping. Two places are required, one as an input place for the antecedent, enabled by default (marking of 1), and another one serving as an input place for the consequent, which will also inhibit t_{sink}, avoiding a violation of the rule.

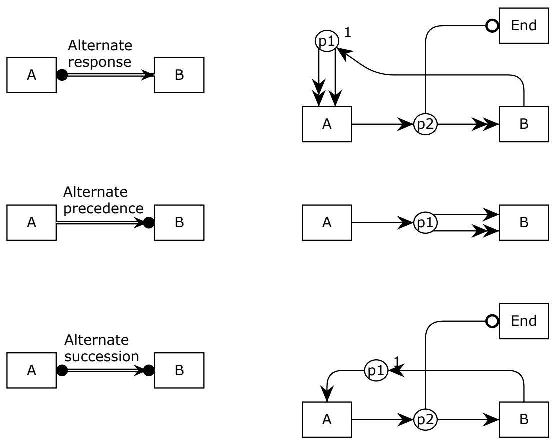

Fig. 5. The mapping of the alternating ordered constraints *Alternate response, precedence*, and *succession*. *Start* is omitted, but provides the initial marking of the helper places (e.g. one token in p_1 for *Alternate response*).

Chain Ordered Constraints. The chain constraints (shown in Figure 6) extensively make use of the *next* LTL operator. To incorporate the setup in which whenever the antecedent is fired, the consequent needs to be next, a helper place is introduced which inhibits every other activity in the net but the consequent. The other activities (besides the antecedent) are indicated in Figure 6 as *C*. The chain precedence constraint in its turn requires a helper place that inhibits the execution of the consequent, which can only fire after the antecedent, which frees the inhibiting place with a reset arc and gets filled by any other activity. The combination of both constraints allows only the execution of the antecedent, and afterwards inhibits every other activity but the consequent, resulting in a strict *ABAB...* pattern.

Negative Constraints. The negative constraints (shown in Figure 7) also use inhibitor places to model violation. *Not co-existence* uses two helper places, one for each activity, which, when containing a marking, block the execution of the other activity. *Not succession* uses one place to inhibit the consequent from executing after the antecedent is ever fired. *Not chain succession* inhibits the consequent from happening exactly after the execution of the antecedent. Again, all the other activities need to be connected to the inhibiting place with a reset arc, freeing the consequent from the marking imposed by the antecedent.

Choice Constraints. The simple *Choice* template inhibits t_{sink} as long as not one of the two involved activities have fired, modeled with two reset arcs, linking both activities to the place. The *Exclusive choice*, is similar to the *Not co-existence* constraint, but inhibits t_{sink} from firing as long as not one of the two

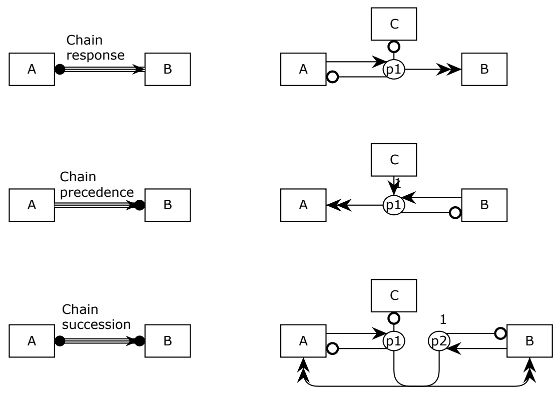

Fig. 6. The mapping of the chain ordered constraints *Chain response, precedence*, and *succession.*

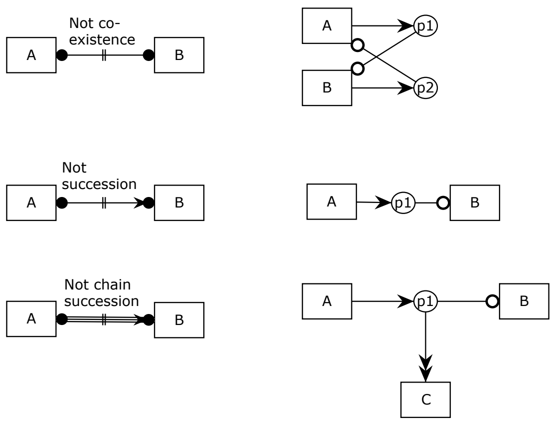

Fig. 7. The mapping of the negative constraints *Not co-existence, succession*, and *chain succession.*

(which is the only possibility) has fired. The helper places in both constraints are marked with a token by default, and are linked to t_{source}. The constraints are shown in Figure 8.

Merging Constraints into a Model. As indicated previously and as shown in the example model (Figure 1), it is possible to merge a set of constraints, having converted these to separate Petri net templates, into one single model as follows. First, note that since the constraints are modeled as separate regions, the overlap of places is void. Second, transitions, are merged so that the input and output arcs for each transition forms the union of all input and output arcs in the separate templates respectively. Third, activity transitions that do not have an input place with corresponding input arc (not a reset or inhibitor arc)

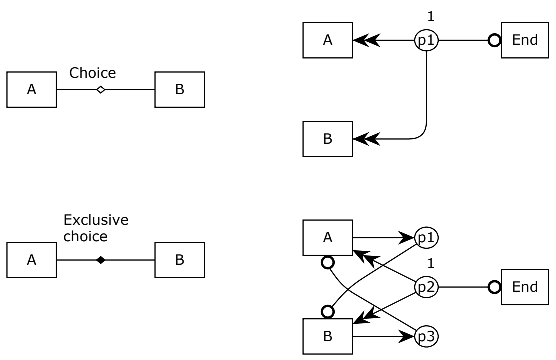

Fig. 8. The mapping of the choice constraints *Choice*, and *Exclusive choice*.

after performing the steps above, are connected with t_{source} with an extra place, which also keeps these transitions enabled after firing by means of a self-loop (i.e. an arc from the transition to this extra place is also added). When t_{sink} does not have an input place, the same rule applies (i.e. also connected with t_{source}, but without a self-loop. Fourth, a unique source and sink place, p_{source} and p_{sink}, are added, and every place in the model except p_{source} and p_{sink} are reset by firing t_{sink} (by adding reset arcs). Note that these steps *also need to be performed when only dealing with the conversion of one single Declare constraint.* For the sake of clarity, however, the adding of these extra constructs is not shown in the separate constraints listed above, but is done in the example model in Figure 1.

An additional example of how a sole constraint fits into a model is given in Figure 9. The constructs used for the *Response(A,B)* constraint are depicted in black and are supplemented by p_0, t_{source}, p_A and p_B which act as the self-loops places of A and B, and the input place p_{E_1} and output place p_{E_2} of *End*. These constructs are indicated in gray and are only used once, even when merging multiple constraints between A and B, while the constructs indicated in black are specific to *Response(A,B)*. Two reset arcs are connected to the self-loop places to empty the net after firing t_{sink}.

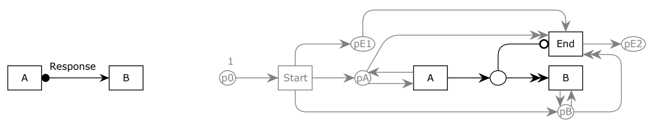

Fig. 9. Example of the *Response* constraint in a model. The constructs specific to the constraint are in black, the ones used for the model are in gray.

n-Ary and Target-Branched Declare Constraints. Some Declare constraints, such as *Choice*, have multiple variants such as *Choice1of3(A,B,C)*, *Choice2of3(A,B,C)*, etc. These variants can be easily expressed in R/I-nets as well. One has to simply add multiple transitions to p_1 in Figure 8, one for every activity. *Choice2of3* can be modeled with two places, etc.

Target-branched constraints, Declare constraints for which the consequent consists of a set, can also be modeled with R/I-nets. We do not elaborate on this for every constraint but rather give an example in the form of *Response(A,B,C)*. The addition of C would simply require an extra reset arc from p_1 to t_C.

4 Equivalence Analysis

4.1 State Spaces and Automata

A Petri net's reachability graph, which is an exhaustive enumeration of all states in the net, can be defined as a Kripke structure. We define such structures as follows. A Kripke structure is a tuple $KS = (S, I, R, L)$ with

- S a finite set of states
- $I \subseteq S$ a set of initial states
- $R \subseteq S \times S$ the transition relations which are left-total ($\forall s \in S\ \exists\ s'$ such that $(s, s') \in R$)
- L a labeling function $L : S \to \mathbb{P}(AP)$ for alphabet AP.

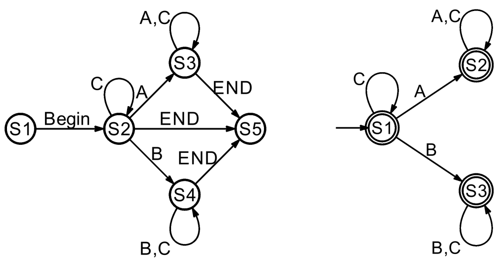

Fig. 10. To the left, the state space for the *Not co-existence* Petri net template is depicted with $S = \{S1, S2, S3, S4, S5\}$, $AP = \{Begin, A, B, C, END\}$, $R = \{(S1, S2), (S2, S3), (S2, S4), (S3, S5), (S4, S5)\}$, $L = \{(S1, \{BEGIN\}), (S2, \{A, B, C, END\}), (S3, \{A, C, END\}), (S4, \{B, C, END\}),\}$, and $I = \{S1\}$. To the right, the equivalent FSA is given.

Translated to a Petri net, every state $s \in S$ is a marking in the net, $S = M$, with an outgoing arc for every enabled transition in that marking. Hence $AP = \Sigma_{PN}$, $L : M \to \mathbb{P}(\Sigma_{PN})$. Since we use a dedicated source place and transition, $I = M_0$ with $M_0(p_{source}) = 1$. We use C to represent all other activities $\Sigma \setminus \{A(,B), \lambda_{Start}, \lambda_{End}\}$, of which the latter are indicated as *Begin* and *End*.

In order to match this with Declare automata, we convert the reachability graph of every Petri net to a finite state automaton. Since the constraints yield regular languages and are closed for certain properties, this conversion yields a regular language model when combining constraints as well. The automaton is defined as a tuple, $FSA = (Q, \Sigma, Q_0, \Delta, A)$ with:

- Q the finite set of states
- Σ the finite alphabet
- $Q_0 \subseteq Q$ the initial states
- $\Delta \subseteq Q \times \sigma \times Q$ the transition relations
- $A \subseteq Q$ a set of accepting states.

Then:

- $Q = M \cup I$
- $Q_0 = I$
- $\Sigma = \Sigma_{PN}$
- $\forall s, s' \in Q, a \in \Sigma : (s, a, s') \in \Delta \iff L(s') = a \wedge (((s, s') \in R) \vee (s = s^i \wedge s' = s^0))$
- $\forall a \in A, \lambda_{End} \in a.$

This can be done for every Petri net template included in the previous conversion and can also be done for Büchi automata, used in [25]. An example is included for the *Not co-existence* template in Figure 10, which displays the state-space to the left, and the corresponding finite state automaton to the right. Every automaton includes all the accepting and temporarily violated states in the Declare automata, as for example included in [38]. Hence, the Petri nets' languages are equal to the Declare automata's languages. The other templates are included below in Figures 11 to 17.

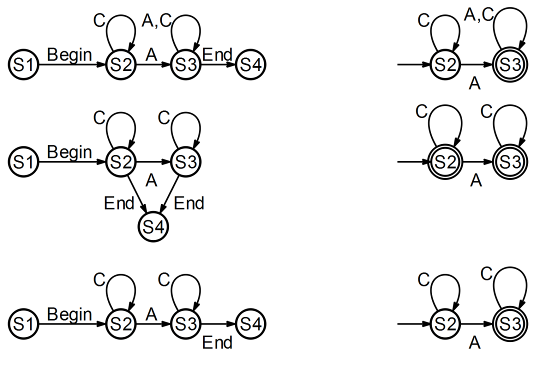

Fig. 11. *Existence1, Absence*, and *Exactly1* constraints.

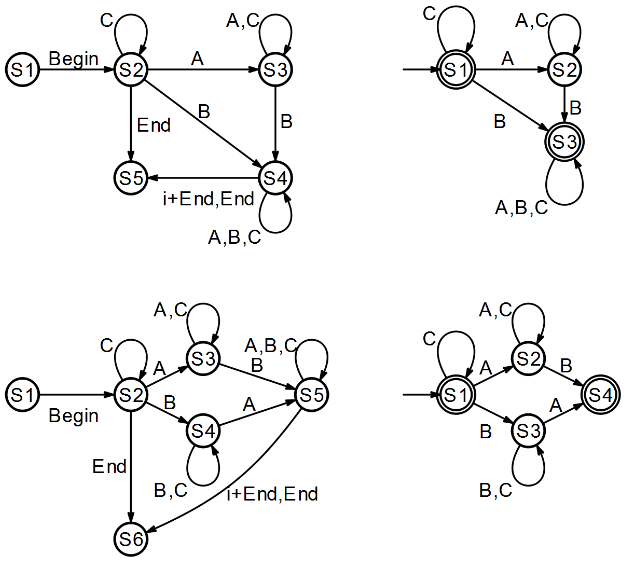

Fig. 12. *Responded*, and *Co-existence* constraints.

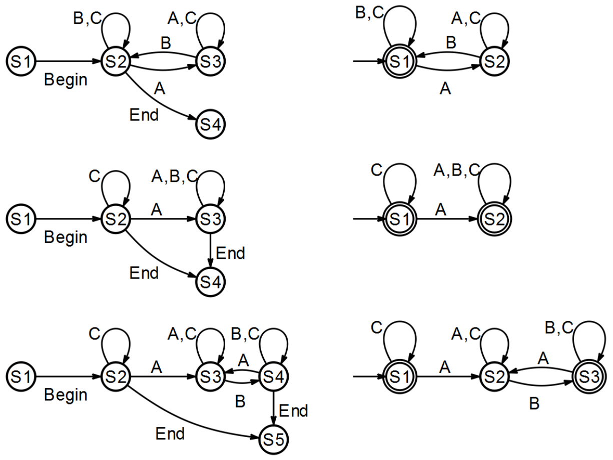

Fig. 13. *Response, Precedence*, and *Succession* constraints.

4.2 Empirical Validation by Simulation

We have discussed the validity of the conversion in the previous section. We now show how the conversions can be used in a practical setting by means of simulation as follows. First we simulate traces from all the different Petri net templates separately and mine them with a Declare process miner (traces generated from R/I-net mined to Declare model), next we simulate traces from

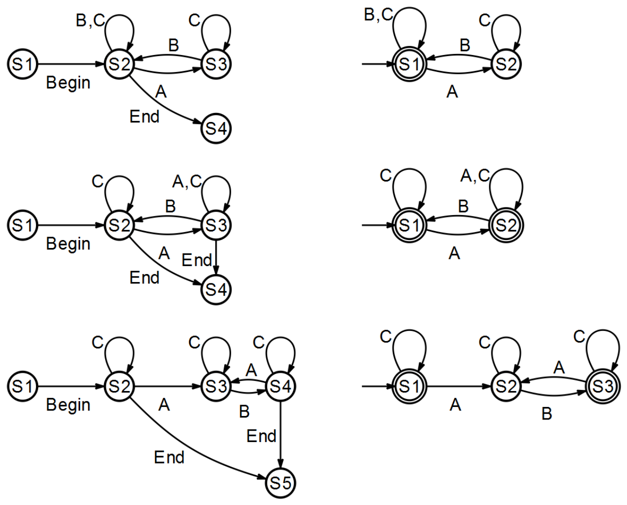

Fig. 14. *Alternate response, precedence*, and *succession* constraints.

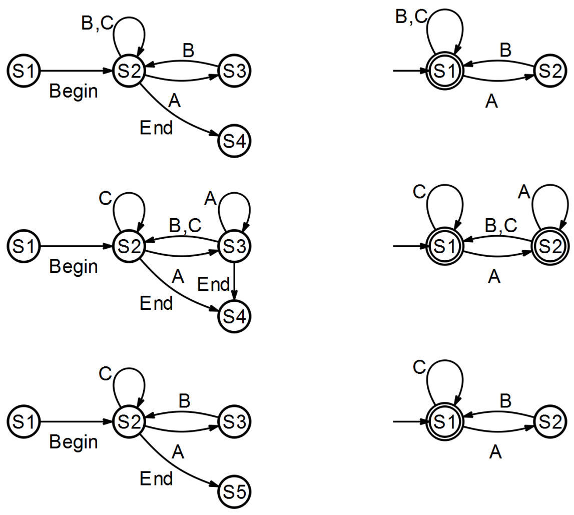

Fig. 15. *Chain response, precedence*, and *succession* constraints.

different Declare constraints and replay them over the corresponding Petri net conversions (direction: Declare model traces replayed over Petri net). We assume that the finite state and Büchi automata yield the same language over finite traces.

The basic setup for the first direction (traces generated from R/I-net mined to Declare model) is performed by constructing a regular Petri net with a simple sequential process placed in a loop, which is then supplemented by one or two

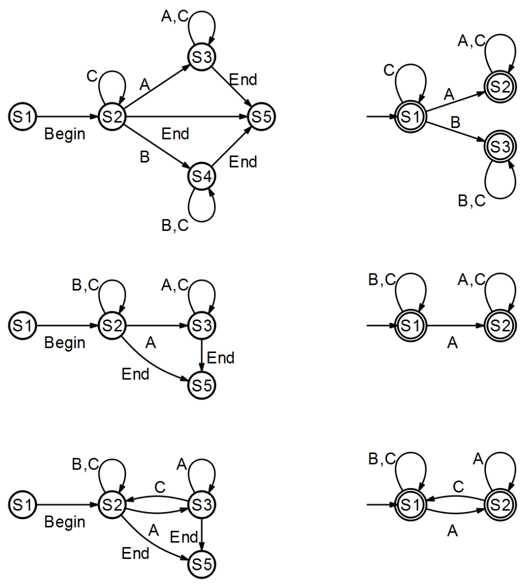

Fig. 16. *Not co-existence, Not succession*, and *Not chain succession* constraints.

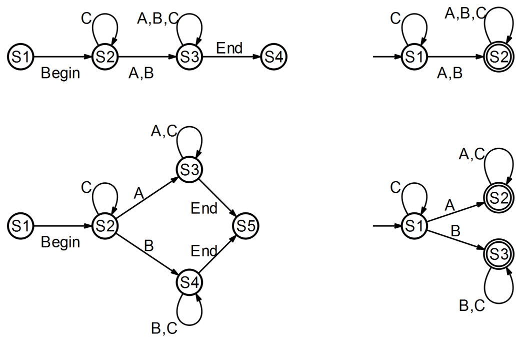

Fig. 17. *Choice*, and *Exclusive choice* constraints.

activities (unary vs. binary) that are constrained by a chosen Petri net template following a Declare conversion. The same considerations are used to create the model, i.e., t_{source}, t_{sink} and self-loops are added. CPN Tools [35] is used to produce models, simulate them, and create an event log, similar to [15]. An example for simulating *Alternate response(B,Z)* is given in Figure 18, in which a straightforward loop is supplemented with the matching Petri net template. The places used for the self-loops input the trace ID with a colored token and the trace identifier is incremented every time t_{sink} is fired. Every simulated event log contains 100.000 traces. The resulting log is then mined with Declare

Miner[1] [24] afterwards. We expect that, for each of the Petri net templates, the corresponding Declare constraint was found by the miner to be either present and supported 100%, or exhibiting a confidence of 100% [24], thus confirming the validity of our technique for this direction.

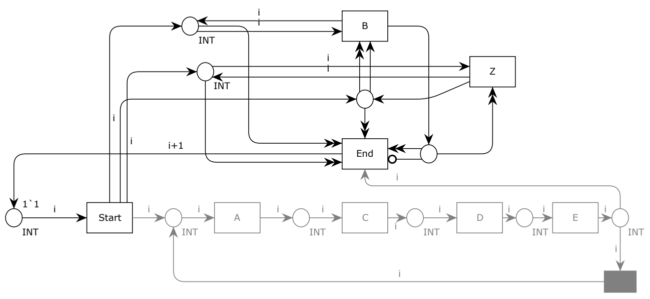

Fig. 18. Example of the verification simulation for *Alternate Response(B,Z)*. The transition actions for writing the log are omitted for clarity. Note that, again, *Start* and *End* serve the purpose of t_{source} and t_{sink} respectively, while also supporting the simulation by initiating and ending a new trace.

Next, we simulate the other way around (Declare model traces replayed over Petri net). A regular expression simulator is used, which uses the expressions in Table 1, in which the . is substituted by C. The simulated traces are then replayed over the Petri net conversions in ProM using the conformance checking plugin used in [8]. When no tokens need to be inserted, the net is not too restrictive for discovering a certain Declare constraint.

Both simulation strategies yield positive results. Declare Miner indeed reported a confidence of 100% for every process log generated from the conversions, making sure that they are at least as strict as their Declare counterparts. The replaying algorithm could fittingly execute every single trace generated for the Declare constraints over the Petri net conversions, proving that the conversions are not looser (resulting in more behavior) than their Declare counterparts. The simulated Declare constraints could be replayed over the Petri net conversions without violation and token inserts, which means the conversions are not stricter than their regular expression counterparts. All the data generated and models used for the simulation have been made available online[2]. They are pluggable and can be used in different settings.

[1] Declare Miner is available in the ProM process mining framework, see: http://www.processmining.org/

[2] http://j.processmining.be/dec2pet

5 Conclusion

This paper presents a template-based conversion of every Declare constraint into a single weighted R/I-net. As such, a formalization of the execution semantics of Declare is obtained, similar to LTL or regular expressions, but now expressed in the form of Petri nets. Equivalence of Declare constraints and the respective Petri net templates is analyzed at the theoretical level and with a simulation experiment, which shows that the Petri net templates are at least as strict as their Declare constraint counterpart, while the templates are not stricter than the regular expressions that go with each Declare constraint.

References

1. van der Aalst, W.M.P.: Formalization and verification of event-driven process chains. Information & Software Technology 41(10), 639–650 (1999)
2. van der Aalst, W.M.P.: Making work flow: On the application of petri nets to business process management. In: Applications and Theory of Petri Nets 2002, 23rd International Conference, ICATPN 2002, Adelaide, Australia, June 24-30, 2002, Proceedings. pp. 1–22 (2002)
3. van der Aalst, W.M.P., Adams, M., ter Hofstede, A.H.M., Pesic, M., Schonenberg, H.: Flexibility as a service. In: Database Systems for Advanced Applications April 20-23, 2009. pp. 319–333 (2009)
4. van der Aalst, W.M.P., ter Hofstede, A.H.M.: YAWL: yet another workflow language. Inf. Syst. 30(4), 245–275 (2005)
5. van der Aalst, W.M.P., Pesic, M.: Decserflow: Towards a truly declarative service flow language. In: The Role of Business Processes in Service Oriented Architectures, 16.07. - 21.07.2006 (2006)
6. van der Aalst, W.M., van Hee, K.M., ter Hofstede, A.H., Sidorova, N., Verbeek, H., Voorhoeve, M., Wynn, M.T.: Soundness of workflow nets: classification, decidability, and analysis. Formal Aspects of Computing 23(3), 333–363 (2011)
7. Adams, M., Ter Hofstede, A.H., Edmond, D., van der Aalst, W.M.: Worklets: A service-oriented implementation of dynamic flexibility in workflows. In: On the Move to Meaningful Internet Systems 2006: CoopIS, DOA, GADA, and ODBASE, pp. 291–308. Springer (2006)
8. Adriansyah, A.: Replay a log on petri net for performance/conformance plug-in. Tech. rep., Technische Universiteit Eindhoven (2012)
9. Adriansyah, A., van Dongen, B.F., van der Aalst, W.M.: Conformance checking using cost-based fitness analysis. In: Enterprise Distributed Object Computing Conference (EDOC), 2011 15th IEEE International. pp. 55–64. IEEE (2011)
10. vanden Broucke, S., De Weerdt, J., Vanthienen, J., Baesens, B., et al.: Determining process model precision and generalization with weighted artificial negative events. IEEE Transactions on Knowledge and Data Engineering 26(8), 1877–1889 (2013)
11. Chinosi, M., Trombetta, A.: BPMN: an introduction to the standard. Computer Standards & Interfaces 34(1), 124–134 (2012)
12. Cortadella, J., Kishinevsky, M., Lavagno, L., Yakovlev, A.: Deriving petri nets from finite transition systems. Computers, IEEE Transactions on 47(8), 859–882 (1998)
13. Couvreur, J.M.: On-the-fly verification of linear temporal logic. In: Formal Methods, pp. 253–271. Springer (1999)

14. De Giacomo, G., Masellis, R.D., Montali, M.: Reasoning on LTL on finite traces: Insensitivity to infiniteness. In: Proceedings of the Twenty-Eighth AAAI Conference on Artificial Intelligence, July 27 -31, 2014, Québec City, Québec, Canada. pp. 1027–1033 (2014)
15. De Medeiros, A.A., Günther, C.W.: Process mining: Using CPN tools to create test logs for mining algorithms. In: Proceedings of the sixth workshop on the practical use of coloured Petri nets and CPN tools (CPN 2005). vol. 576 (2005)
16. Di Ciccio, C., Mecella, M.: A two-step fast algorithm for the automated discovery of declarative workflows. In: Computational Intelligence and Data Mining (CIDM), 2013 IEEE Symposium on. pp. 135–142. IEEE (2013)
17. Dumas, M., La Rosa, M., Mendling, J., Reijers, H.A.: Fundamentals of business process management. Springer (2013)
18. Fahland, D.: Towards analyzing declarative workflows. In: Autonomous and Adaptive Web Services, 04.02. - 09.02.2007 (2007)
19. Fahland, D., Lübke, D., Mendling, J., Reijers, H., Weber, B., Weidlich, M., Zugal, S.: Declarative versus imperative process modeling languages: The issue of understandability. In: Enterprise, Business-Process and Information Systems Modeling, pp. 353–366. Springer (2009)
20. Goedertier, S., Vanthienen, J., Caron, F.: Declarative business process modelling: principles and modelling languages. Enterprise IS 9(2), 161–185 (2015)
21. Hack, M.: Petri net language. Massachusetts Institute of Technology (1976)
22. Kumar, A., Yao, W.: Process materialization using templates and rules to design flexible process models. In: Rule Interchange and Applications, pp. 122–136. Springer (2009)
23. de Leoni, M., Maggi, F.M., van der Aalst, W.M.: An alignment-based framework to check the conformance of declarative process models and to preprocess event-log data. Information Systems (2014)
24. Maggi, F.M., Mooij, A.J., van der Aalst, W.M.: User-guided discovery of declarative process models. In: Computational Intelligence and Data Mining (CIDM), 2011 IEEE Symposium on. pp. 192–199. IEEE (2011)
25. Maggi, F.M., Westergaard, M., Montali, M., van der Aalst, W.M.: Runtime verification of ltl-based declarative process models. In: Runtime Verification. pp. 131–146. Springer (2012)
26. de Medeiros, A.K.A., Weijters, A.J., van der Aalst, W.M.: Genetic process mining: an experimental evaluation. Data Mining and Knowledge Discovery 14(2), 245–304 (2007)
27. Murata, T.: Petri nets: Properties, analysis and applications. Proceedings of the IEEE 77(4), 541–580 (1989)
28. Pesic, M., Schonenberg, H., van der Aalst, W.M.: Declare: Full support for loosely-structured processes. In: Enterprise Distributed Object Computing Conference, 2007. EDOC 2007. 11th IEEE International. pp. 287–287. IEEE (2007)
29. Pesic, M., van der Aalst, W.M.: A declarative approach for flexible business processes management. In: Business Process Management Workshops. pp. 169–180. Springer (2006)
30. Prescher, J., Ciccio, C.D., Mendling, J.: From declarative processes to imperative models. In: Proceedings of the 4th International Symposium on Data-driven Process Discovery and Analysis (SIMPDA 2014), Milan, Italy, November 19-21, 2014. pp. 162–173 (2014)
31. Reijers, H.A., Slaats, T., Stahl, C.: Declarative modeling–an academic dream or the future for bpm? In: Business Process Management, pp. 307–322. Springer (2013)

32. Sadiq, S., Sadiq, W., Orlowska, M.: Pockets of flexibility in workflow specification. In: Conceptual ModelingER 2001, pp. 513–526. Springer (2001)
33. Somenzi, F., Bloem, R.: Efficient büchi automata from ltl formulae. In: Computer Aided Verification. pp. 248–263. Springer (2000)
34. Verbeek, H., Wynn, M.T., van der Aalst, W.M., ter Hofstede, A.H.: Reduction rules for reset/inhibitor nets. Journal of Computer and System Sciences 76(2), 125–143 (2010)
35. Westergaard, M.: Cpn tools 4: multi-formalism and extensibility. In: Application and Theory of Petri Nets and Concurrency, pp. 400–409. Springer (2013)
36. Westergaard, M., Maggi, F.M.: Declare: A tool suite for declarative workflow modeling and enactment. BPM (Demos) 820 (2011)
37. Westergaard, M., Slaats, T.: Mixing paradigms for more comprehensible models. In: Business Process Management, pp. 283–290. Springer (2013)
38. Westergaard, M., Stahl, C., Reijers, H.A.: Unconstrainedminer: Efficient discovery of generalized declarative process models. Tech. Rep. BPM-13-28, BPMcenter (2013)

FACULTY OF ECONOMICS AND BUSINESS
Naamsestraat 69 bus 3500
3000 LEUVEN, BELGIË
tel. + 32 16 32 66 12
fax + 32 16 32 67 91
info@econ.kuleuven.be
www.econ.kuleuven.be