Improving Understandability of Declarative Process Models by Revealing Hidden Dependencies

Johannes De Smedt(✉), Jochen De Weerdt, Estefanía Serral, and Jan Vanthienen

Department of Decision Sciences and Information Management, Faculty of Economics and Business, KU Leuven, Naamsestraat 69, 3000 Leuven, Belgium
{johannes.desmedt,jochen.deweerdt,estefania.serral, jan.vanthienen}@kuleuven.be

Abstract. Declarative process models have become a mature alternative to procedural ones. Instead of focusing on what has to happen, they rather follow an outside-in approach based on a rule base containing different types of constraints. The models are well-capable of representing flexible behavior, as everything that is not forbidden by the constraints in the model is possible during execution. These models, however, are more difficult to comprehend and require a higher mental effort of both the modeler and the reader. Since constraints can be added freely to the model, it is often overseen what impact the combination of them has. This is often referred to as hidden dependencies. This paper proposes a methodology to make these dependencies explicit for the declarative process modeling language Declare by considering a Declare model as a graph and relying on the constraints' characteristics. Moreover, this paper also contributes by empirically confirming that a tool that can visualize hidden dependency information on top of a Declare model has a significant positive impact on the understandability of Declare models.

Keywords: Declarative process modeling · Declare · Hidden dependencies · Empirical evaluation

1 Introduction

Declarative process models have been proposed to counter the flexibility limitations of procedural modeling languages. Instead of modeling predetermined paths of activities, declarative process models use constraints to express what can, cannot, and must happen. Every execution sequence that is not strictly forbidden by the constraints can be enacted by the model. This makes declarative models much more flexible indeed, but also more difficult to comprehend. To put it simplistically, it is not possible to 'find an execution path by following your finger along the arcs'. There are many possible outcomes due to the interaction of the constraints over the activities.

© Springer International Publishing Switzerland 2016
S. Nurcan et al. (Eds.): CAiSE 2016, LNCS 9694, pp. 83–98, 2016.
DOI: 10.1007/978-3-319-39696-5_6

In different works approaches to deal with the understandability problems of declarative models have been proposed. For instance in [1], the impact of hierarchy is investigated and in [2] the typical pitfalls of understanding declarative models are pointed out.

This paper proposes an approach capable of improving the understandability of models expressed in Declare [3], one of the most widely used declarative process modeling language. The approach deals with hidden dependencies [2,4,5], one of the main reasons that make Declare models difficult to understand. Hidden dependencies pose a significant challenge for humans: it is not sufficient to rely on the information explicitly indicated by the constraints, but one has to carefully analyze all the defined constraints for understanding all the dependencies that are not explicitly visible (i.e., that are hidden). The contribution of this paper consists of a methodology to build so-called constraint dependency structures in order to reveal all hidden dependencies and make them explicit in a Declare model. Furthermore, this methodology is developed into the Declare Execution Environment[1], a tool that supplements an existing Declare model with visual and textual annotations to clarify which behavior is allowed or disallowed by the model. In an experimental evaluation with 95 novice Declare modelers, we show that the methodology to make hidden dependencies explicit and visually annotating a Declare model with this information, has a significant positive impact on the understandability of Declare models.

The structure of the paper is as follows. First, the concept of Declare constraints is briefly summarized and relevant characteristics are explained. Next, Sect. 3 explains how to capture and formalize dependency structures, followed by Sect. 4, which shows the implementation and tool. This tool is used for experimental validation in Sect. 5. Finally, Sect. 6 summarizes the related work and Sect. 7 discusses future work and the conclusion.

2 Declare Constraints and Their Characteristics

Declare models are constructed using a fixed set of constraints, which are summarized in Table A1 in [6]. They range from unary constraints, indicating the position and cardinality of an activity, to n-ary constraints, which capture typical sequential behavior such as precedence and succession relationships. A Declare model $DM = (A, \Pi)$ can be represented as follows:

- A is a set of activities from the alphabet Σ,
- Π is the set of Declare constraints defined over the activities.

In this paper we assume $n \leq 2$. A Declare graph can be represented as a directed graph $DG = (A, \Pi)$. Hence the activities and constraints map one-to-one onto the graph in case of $n = 2$, given that unary constraints are considered as self-loops. We denote all incoming arcs of $a \in A$ as $\bullet a \subseteq \Pi$ and outgoing arcs as $a\bullet \subseteq \Pi$. The antecedent and consequent of $\pi \in \Pi$ are denoted as π_a and π_b.[2]

[1] http://www.processmining.be/declareexecutionenvironment.

[2] Our interpretation of these concepts differ from, e.g., [7], as we consider a the antecedent in $Precedence(a, b)$ relationships, rather than b for notational simplicity.

The execution of a Declare model can be realized by constructing an automaton (either a Büchi [8] or finite state automaton [9,10]) by conjoining the different constraints' automata to obtain the behavior that is allowed for by all of them. This conjunction actually abolishes the notion of the separate constraints and thus throws away the information of how the separate constraints interact. One technique to mitigate this is to color the constraints [11] by keeping both the global and separate automata, but still the interactions remain untraceable.

Declare constraints exhibit a hierarchy, which is well-explained in [8,9]. For unary constraints, *Existence(A,n)* and *Absence(A,n)* together form *Exactly (A,n)*. Binary constraints are divided in different classes, for which every class depends on the previous one: *Unordered (Responded/co-existence)*, *Simple ordered (Precedence (p), Response (r), Succession (s))*, *Alternating ordered (Alternate p,r,s)*, and *Chain Ordered (Chain p,r,s)*. Next to these constraints, there exist negative versions for three of them *(Not co-existence, Not succession, Not chain succession)*. Finally, the *Choice* constraint exists, which is comparable with a branched unary constraint *Existence(*$\{A,B\}$*,n)*.

For binary constraints, *(Alternate/Chain) Response(A,B)* and *(Alternate/Chain) Precedence(A,B)* form *(Alternate/Chain) Succession*. When a property is discussed for, e.g., *Chain succession*, this also includes *(Chain/Alternate) precedence/response* and vice versa.

Furthermore, each constraint has specific characteristics that are discussed in [12]. Some constraints have an impact on the temporary violation aspect of the model (the constraint is not in an accepting state and requires an activity to resolve it, e.g. *Response* or *Choice*), some constraints can disable activities for the remainder of the execution (such as *Exactly* and *Not succession*), and some constraints can temporarily block all other activities (*Chain* constraints). These different characteristics all impose certain dependencies among constraints that are not directly visible through a single constraint (arc). E.g., a model consisting of $A = \{a, b, c\}$ and $\Pi = \{Response(a, b), Response(b, c), Exactly(c, 2)\}$ contains a hidden dependency between a and c. When c is fired once (and hence can only fire one time anymore), and a has fired without b firing already, c should not fire before b resolves the temporary violation of $Response(a, b)$, since after firing c, c cannot resolve $Response(b, c)$ anymore (as it can only fire two times) and b should not fire to avoid another temporary violation of $Response(b, c)$.

The hidden dependencies caused by all these characteristics can be made explicit. In the following section, it is explained how they relate to the activities in a declarative model.

3 Declare Dependency Structures

This section discusses how dependency structures retrieved from Declare models can be constructed (Sect. 3.1), how they can aid interpretation of the model and the way in which constraints interact (Sect. 3.3). Before constructing the structures, however, the unary constraints in the model need to be propagated to achieve the correct interpretation, as explained in Sect. 3.2.

3.1 Construction

A hidden dependency can be defined as an interaction between constraints and their activities that is not made explicit as such in the model itself. They are the outcome of conjoining the separate constraints to avoid permanent violation, as explained earlier. Hence, it is paramount to find the ways to avoid permanent violation to occur. There are three types of resolution strategies to resolve temporary violations:

1. **An activity must still happen:** after firing the antecedent in *Responded existence*, *Co-Existence*, *(Alternate/Chain) Response*, the consequent must fire afterwards.
2. **An activity must still happen a certain amount of times:** *Existence*, *Exactly*, *Choice*.
3. **An activity must still happen at a fixed moment in time:** *Chain response*.

Note that combining different constraints could lead to coalesced resolution strategies: *Chain response(a,b)* coupled with *Existence(a,2)* requires firing b at least twice on certain fixed moments (directly after a) as well.

Now we construct the set of dependency structures DP for DM with $DS = (\pi^{DS}, \Pi^{DS}_{dep}, DS^{DS}_{dep}), DS \in DP$ with

- π^{DS} the constraint triggering the structure,
- Π^{DS}_{dep} the set of dependent constraints, and
- DS^{DS}_{dep} the set of nested dependency structures dependent of π^{DS}.

To fill Π^{DS}_{dep} and DS^{DS}_{dep}, Algorithm 1 creates a dependency structure for every activity that is involved in at least one of the five constraints that can permanently disable it. Hence, a structure is created for a in *Absence/Exactly(a,n)*, a and b in *Exclusive choice/Not co-existence(a,b)*, and for b in *Not succession(a,b)* as can be seen on lines 7–25.

First, all backward-propagating constraints are considered ($\Pi_{BW} \subseteq \Pi$, inferred from resolution strategy 1) and used for recursive search, as well as stored in Π_{dep} (Algorithm 2, lines 1–22). During this procedure, all incoming *Existence* and *Choice* constraints (as in 2) are stored as well (Algorithm 2, lines 16–18). They also need to be fulfilled, but do not propagate due to their unary nature. When *Responded existence* is encountered, a new dependency structure $DL \in DS^{DS}_{dep}$ is constructed because when the constraint becomes satisfied (by firing its consequent), it is satisfied indefinitely (unlike, e.g., *Response* which can become temporarily violated again) and its propagation is also abolished (Algorithm 2, lines 6–10).

For every activity that is encountered by the algorithm, a forward-dependency search is performed for all forward-propagating constraints $\Pi_{FW} \subseteq \Pi$, which include all (*Alternate/Chain*) *precedence* constraints and *Co-existence*. These constraints need to be activated (the antecedent has to be fired, in the case of alternating variant possibly multiple times) to resolve dependencies

Algorithm 1. Retrieving Dependency Structures

Input: $DM = (A, \Pi)$
Input: $\Pi_{BW} \leftarrow \Pi_{Resp/CoEx} \cup \Pi_{(C/A)Response}$ ▷ Backward-propagating constraints
Input: $\Pi_{FW} \leftarrow \Pi_{CoEx} \cup \Pi_{(C/A)Precedence}$ ▷ Forward-propagating constraints
Output: DP ▷ The set of dependency structures for DM

1: **procedure** ReturnDepTrans(DM)
2: $DP \leftarrow \emptyset$ ▷ The set of all dependency structures of the model
3: **for** $\pi \in \Pi$ **do**
4: $DS \leftarrow \emptyset$ ▷ The dependent structure for π
5: $V^l \leftarrow \emptyset$ ▷ Set of visited activities for left search
6: $V^r \leftarrow \emptyset$ ▷ Set of visited activities for right search
7: **if** $\pi \in \Pi_{Abs} \vee \pi \in \Pi_{Exa}$ **then**
8: $\pi^{DS} \leftarrow \pi$
9: $DS \leftarrow SearchLeft(\pi_a, V^l, DS) \cup SearchRight(\pi_a, V^r, DS)$
10: $DP \leftarrow DS$
11: **end if**
12: **if** $\pi \in \Pi_{NotSuc}$ **then**
13: $\pi^{DS} \leftarrow \pi$
14: $DS \leftarrow SearchLeft(\pi_b, V^l, DS) \cup SearchRight(\pi_b, V^r, DS)$
15: $DP \leftarrow DS$
16: **end if**
17: **if** $\pi \in \Pi_{ExclChoi} \vee \pi \in \Pi_{NotCoEx}$ **then**
18: $\pi^{DS} \leftarrow \pi$
19: $DS \leftarrow SearchLeft(\pi_a, V^l, DS) \cup SearchRight(\pi_a, V^r, DS)$
20: $DP \leftarrow DS$
21: $DS_2 \leftarrow \emptyset$
22: $\pi^{DS_2} \leftarrow \pi$
23: $DS_2 \leftarrow SearchLeft(\pi_b, V^l, DS) \cup SearchRight(\pi_b, V^r, DS_2)$
24: $DP \leftarrow DS_2$
25: **end if**
26: **end for**
27: **return** DP
28: **end procedure**

from backward-propagating constraints. The constraints dependent of them are linked to them through a separate, nested dependency structure $DL \in DS^{DS}_{dep}$ (Algorithm 2, lines 22–36).

Example. Consider the model in Fig. 1a. *Not succession(c,b)*, meaning any occurrence of c cannot be followed eventually by b, causes the algorithm to construct a dependency structure for b. Backward-searching will reveal *Response(a,b)* and *Exactly(a,1)* as dependent constraints. c cannot fire before a has resolved *Exactly(a,1)*, which will render *Response(a,b)* temporarily violated and requires b to resolve it. Hence sequences such as $\sigma = e \rightarrow b$ or $\sigma = a \rightarrow e$ are not possible. In a forward search, *Precedence(b,d)* requires a new dependency structure, nested in $DS_{Not\_succession(C,B)}$. Firing e requires d to resolve *Response(e,d)*. Hence, firing c before firing e would render e disabled, as b can never fire anymore due to *Not succession(c,b)*, so the *Precedence(b,d)* can never be activated. Firing b before c would resolve this, as d can then fire an unlimited amount of times. The full dependency structure present in the model is $DS = \{\pi = Not\_succession(c,b), \Pi_{dep} = \{Response(a,b), Exactly(a,1)\}, DS_{dep} = \{\pi = Precedence(b,d), \Pi_{dep} = Response(e,d), \emptyset\}\}$.

Algorithm 2. Search for Dependency Constraints

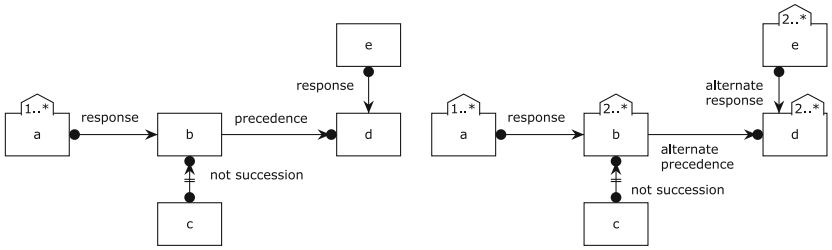
procedure SEARCHLEFT(a, V, DS)
  if $\neg(a \in V)$ **then** ▷ Do if a is not visited yet, avoids infinite loops
    $V \leftarrow a$
    for $\pi \in \bullet a$ **do** ▷ Scan all incoming Declare constraints of activity a
      if $\pi \in \Pi_{BW}$ **then**
        if $\pi \in \Pi_{RespEx}$ **then**
          $DL \leftarrow \emptyset$ ▷ Create new nested dependency structure
          $\pi^{DL} \leftarrow \pi$
          $DL \leftarrow SearchLeft(\pi_a, V, DL) \cup SearchRight(\pi_a, V, DL)$
          $DS^{DS}_{dep} \leftarrow DL$ ▷ Add nested structure to main structure DS
        else
          $\Pi^{DS}_{dep} \leftarrow \pi$
          $DS \leftarrow SearchLeft(\pi_a, V, DS) \cup SearchRight(\pi_a, V, DS)$
        end if
      end if
      if $\pi \in \Pi_{Exis} \vee \pi \in \Pi_{Exa} \vee \pi \in \Pi_{Choi}$ **then**
        $\Pi^{DS}_{dep} \leftarrow \pi$
      end if
    end for
  end if
  return DS
end procedure

procedure SEARCHRIGHT(a, V, DS)
  if $\neg(a \in V)$ **then**
    $V \leftarrow a$
    for $\pi \in a\bullet$ **do** ▷ Scan all outgoing Declare constraints of activity a
      if $\pi \in \Pi_{FW}$ **then**
        $DL \leftarrow \emptyset$
        $\pi^{DL} \leftarrow \pi$
        $DL \leftarrow SearchLeft(\pi_b, V, DL) \cup SearchRight(\pi_b, V, DL)$
        $DS^{DS}_{dep} \leftarrow DL$
      end if
    end for
  end if
  return DS
end procedure

3.2 Unary Propagation

The construction and use of dependency structures depends on the correct propagation of all unary relations inside of the model. E.g., consider the model in Fig. 1b. Changing *Precedence(b,d)* and *Response(e,d)* to their alternating variant and adding *Existence(e,2)* would require d and hence b to fire at least twice as well. In general, unary constraints that are not present in the original model are added in the following fashion for $a, b \in A$:

- *Responded existence(a,b)*, if a occurs at least or exactly n times, then b has to occur at least once.
- *Co-existence(a,b)*, if a or b occurs at least or exactly n times, then b respectively a has to occur at least once.
- *Response(a,b)*, if a occurs at least or exactly n times, then b has to occur at least once.
- *Precedence(a,b)*, if b occurs at least or exactly n times, then a has to occur at least once.



(a) Simple example with *Not succession* inflicting hidden dependencies.

(b) The same example with unary and alternating constraints.

Fig. 1. An example of a small Declare model with hidden dependencies in two variants.

- *Succession(a,b)*, if a or b occur at least or exactly n times, then the other activity has to occur at least once.
- *Alternate response(a,b)*, if b occurs at most or exactly n times, then a can occur at most n times. If a occurs at least or exactly m times, then b has to occur at least m times.
- *Alternate precedence(a,b)*, if b occurs at least or exactly n times, then a has to occur at least n times.
- *Alternate succession(a,b)*, both a and b have to have the same unary restrictions.
- *Chain response(a,b)*, if a occurs at least or exactly n times, then b has to occur at least n times. If b occurs at most or exactly m times, then a can only occur at most m times.
- *Chain precedence(a,b)*, if a occurs at most or exactly n times, then b can occur at most n times.
- *Chain succession(a,b)*, both a and b have to have the same unary restrictions.

Every unary constraint has a lower bound *Existence(n)* and upper bound *Absence(m)*, and they are combined and replaced by an *Exactly* constraint when $n = m - 1$. These rules are applied to the model until no unary constraint changes anymore. If there would be an activity for $n > m - 1$, this would mean the model would end up in a permanently violated state.

This propagation is done before the model is used in the algorithms in order to have consistent dependency structures. Next, the same procedure is repeated to calculate for each activity how many times it still has to execute in the following execution steps. This helps the dependency structures to recognize whether a certain nested structure can be cast off because it can fire a sufficient amount of times.

Example. Returning to the example, *Existence(e,2)* is propagated to d, yielding an *Existence(d,2)* and next to b yielding *Existence(b,2)*. The minimum amount of occurrences is also calculated and updated throughout the execution per activity. This way, the dependency structures will incorporate the unary constraints into

the model and indicate that c cannot fire before b has fired its appropriate amount of times to enable d and e to fire at least twice. Initially, b, d, and e must execute a minimum of 2 times. b can be disabled after d and e have fired at least once, and b has fired at least two times (once before d fired and once after d fired to grant d another execution because of *Alternate precedence(b,d)*), for example sequence $\sigma_1 = b \rightarrow e \rightarrow d \rightarrow b \rightarrow c$ or $\sigma_2 = e \rightarrow b \rightarrow d \rightarrow b \rightarrow c$. After this execution, d cannot fire until e is fired because they can both fire only once anymore (d because of *Alternate precedence(b,d)* and hence e through *Alternate response(e,d)*) and d has to be able to resolve *Alternate response(e,d)*, e.g. $\sigma_1 \rightarrow e \rightarrow d$ and not $\sigma_1 \rightarrow d \rightarrow e$.

3.3 Interpretation

Constructing dependency structures can already give extra information by displaying them in a graph showing which constraints interact with the main constraint (π^{DS}) in the structure. However, they can be expressed in extra descriptions to annotate the model in order to help understand why constraints are related and what combined impact they have.

First of all, for *Exclusive choice(a,b)* and *Not co-existence(a,b)*, the structures reflect that whenever an activity from either structure is fired (either the one for a or b), the activities in the other structure become disabled permanently. Indeed, firing any activity in the dependency structure of a or b requires them to fire, hence activating *Exclusive choice* or *Not co-existence*. If the structures of a and b share activities, this means the net is not deadlock-free.

Secondly, for *Not succession(a,b)*, a becomes disabled whenever a constraint $\pi \in \Pi^{DS}_{dep}$ is temporarily violated and needs b to resolve it. Also, dependent structures in $d \in DS^{DS}_{dep}$ cannot contain any violations in their Π^{d}_{dep} unless the antecedent of the main constraint $\pi^d \in DS^{d}_{dep}$ is activated and can execute a minimum number of times required (as explained in Sect. 3.2). For unary constraints, *Absence(A,n)* and *Exactly(A,n)*, this applies as well, with the exception that a becomes disabled when a constraint relies upon it to become satisfied again.

Finally, every execution of activities in *Chain* constraints should be checked for executions one step ahead. For each of them, it is calculated whether the consequent is available to fire for *Chain response*, or is the only one available for *Not chain succession* in order to avoid deadlock.

4 Tool Support

The construction of the dependency structures has been implemented in a Declare execution environment, of which the implementation can be found by following the link in the introduction. The tool can read a Declare model saved from Declare Designer [13], which, during execution, is supported by descriptions for the hidden dependencies. A screenshot and an example can be found in Fig. 2.

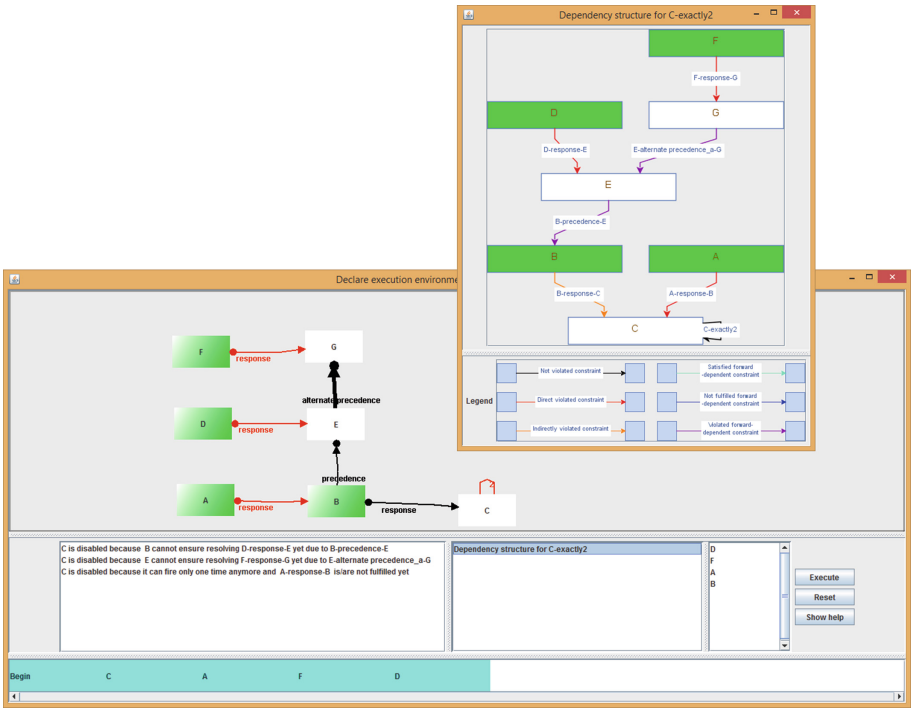


Fig. 2. An example of a small Declare model with hidden dependencies and the corresponding dependency graph for *Exactly(c,2)*.

Furthermore, the dependency structures can be visualized next to the model as a directed graph as well. Finally, the trace created over the model by the user is displayed below the model, aiding the user in understanding the history of the current situation displayed over the model.

The execution semantics are provided by dk.brics.automaton [14] and consists of the conjunction of the separate Declare automata expressed in regular expressions, as can be found in [9,10].

5 Empirical Evaluation

Making hidden dependencies explicit by annotating Declare models can significantly improve their understandability. In this section, it is empirically demonstrated that novice process modelers are indeed better capable of understanding Declare models when they are provided with an environment that makes hidden dependencies explicit through text and figures.

5.1 Experimental Setup

In the experiment, 95 students (see Table 2) enrolled in KU Leuven's *Business Analysis* course, in which both procedural and declarative process modeling are

Table 1. The different Declare models used during the experiments. High-resolution versions of the figures of the models used in the experiment can be found by following the link to the tool site.

Model 1	Model 2	Model 3
Response(a,b)	Exactly(a,2)	Response(a,b)
Precedence(b,c)	Existence(c,2)	Response(b,c)
Not co-existence(b,e)	Exactly(b,2)	Exactly(c,2)
Response(d,e)	Absence(d,3)	Precedence(b,e)
	Alternate precedence(a,c)	Response(d,e)
	Alternate response(b,d)	Alternate precedence(e,g)
		Response(f,g)
Model 4	Model 5	
Response(a,b)	Response(a,b)	Choice(a,j)
Existence(b,1)	Response(b,c)	Not succession(i,j)
Alternate precedence(b,c)	Exactly(c,2)	
Not succession(b,e)	Precedence(b,e)	
Existence(c,1)	Response(d,e)	
Response(d,c)	Alternate precedence(e,g)	
Existence(d,2)	Response(f,g)	

taught, were asked to solve five questions for each of five different Declare models in a timespan of two hours. The students have the same modeling experience and background and can be considered novice business process modelers. The models, as represented in Table 1, are of increasing complexity and are tailored towards assessing different kinds of dependencies:

- **Model 1:** focuses on the impact of the *Not co-existence* constraint.
- **Model 2:** focuses on the impact of unary constraint propagation.
- **Model 3:** focuses on the impact of simple forward and backward dependencies induced by *Exactly(c,2)*.
- **Model 4:** focuses on the impact of more advanced forward and backward dependencies induced by *Not succession(b,e)*.
- **Model 5:** focuses on the same impact as models 3 and 4, with an added *Choice* constraint.

At the start of the test, students were provided instructions making use of the example used in Sect. 2, a model which was used as a foundation for models 3–5, but without the additional constraints and activities added. As such, the idea behind hidden dependencies was explained, as well as how to make use of the tool they were provided with.

In order to measure the impact of handing natural language descriptions and the visualization of dependency graphs, the students were divided into three groups which received a different version of the Declare Execution Environment. Group A could only see the Declare model and the constraint descriptions,

but no color annotation nor dependency structure visualizations. Group B received a tool in which the enabled activities were colored green, and temporarily violated constraints were colored red, in a fashion described in [11] and similar to Declare Designer [13]. Also, the constraint descriptions were given. Finally, group C was given an environment with the same functionality as group B, but with extra descriptions concerning hidden dependencies, as well as the possibility to open a dynamic visualization of the dependency structures.

Table 2. The students were selected from 3 different programs, however, it was made sure their distribution could not skew the results. More info can be found by following the provided link in Sect. 1.

Group	Participants	Gender		Program		
		Male	Female	IS	Business	CS
A	36	25	11	5	31	0
B	32	23	9	6	26	0
C	27	15	12	5	21	1

The questions were aimed at uncovering to which extent the participants grasped the full impact of the blend of different constraints. They were asked to indicate which activities were enabled after firing a certain sequence, and why or how to reach a certain firing sequence. Since two out of three groups knew which ones were enabled, they could focus more on the second part of the question. An example question used for model 1 was 'After firing d, which activities are still enabled? Explain'.

Each question was scored on a 0 to 1 scale, where incomplete answers (usually because of overlooked hidden dependencies or incorrect use of constraints) were still awarded a score higher than 0. E.g., a student from group B who provides the correct set of enabled activities but fails to state that activity c in model 3 is not enabled because of hidden dependencies was still awarded 0.6. The explanation was taken into account so as to make a fair comparison with students in group A, who got no extra information, and therefore many times missed even these basic answers. Group C students that just copied extra descriptions provided by the tool also did no receive a grade of 1, as they did not prove to understand the model.

5.2 Results

Quantitative Results. Given this setup, an experimental analysis can be conducted to investigate the impact of the environment students were given (i.e. group) on the score, where a higher score reflects a better level of understanding. Figure 3 shows boxplots of the average scores over 5 questions, per model and per group. From the figure, it can be seen that for each model, an increase

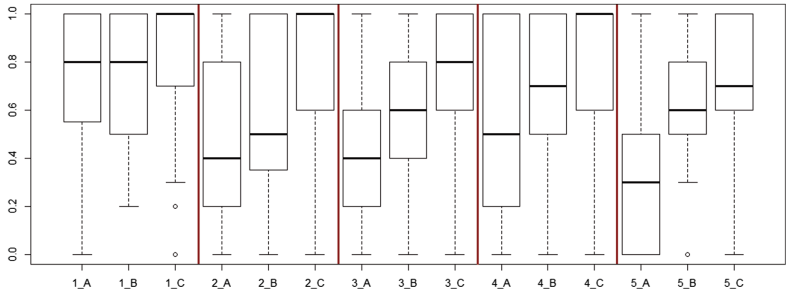


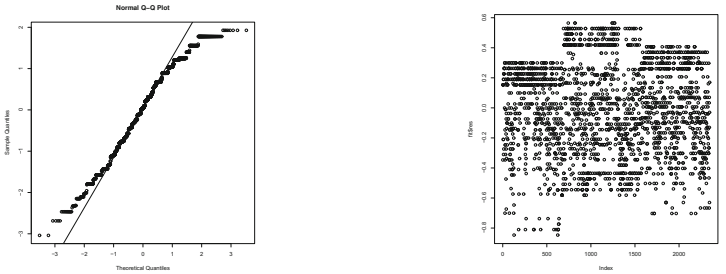
Fig. 3. Boxplot of the scores of 5 questions per model (1–5) and per group (A–C).

Table 3. Linear regression model based on the data gathered from the experiment with significance scores '***' 0, '**' 0.001, and '*' 0.01.

Coefficients	Estimate	Std. errror	t value		Pr(>|t|)	
(Intercept)	0.64092	0.01583	40.498	<	2.00E-16	***
Model2	-0.17082	0.01939	-8.81	<	2.00E-16	***
Model3	-0.17082	0.01939	-8.81	<	2.00E-16	***
Model4	-0.10498	0.01943	-5.403	<	7.22E-08	***
Model5	-0.21555	0.01964	-10.977	<	2.00E-16	***
GroupB	0.15811	0.01463	10.81	<	2.00E-16	***
GroupC	0.26522	0.01524	17.4	<	2.00E-16	***
Residual standard error: 0.2986 on 2340 degrees of freedom						
Multiple R-squared: 0.1658						
Adjusted R-squared: 0.1636						
F-statistic: 77.49 on 6 and 2340 DF, p-value: < 2.2e-16						

is observed in terms of the score when students are provided with additional hidden dependency-based annotations. Note that the data is available on the tool's web site.

So as to evaluate the statistical significance of this pattern, a linear regression ($Score = \alpha \times model + \beta \times group + \epsilon$) was fitted on the data. From the results in Table 3, it is clear that both the impact of the model as well as the group (and hence tool) is highly significant. Observe that the data was also fitted for a model with interaction between *model* and *group* and also for a model with *gender* and *program* included. These models did not raise the R-squared values much (<0.18), hence hinting at little extra explanatory power. Running a Durbin-Watson-test also rejected the hypothesis for correlation among the residuals. Finally, it was tested whether the error terms were normally distributed, as can be seen in Fig. 4a.



(a) Q-Q plot of the error terms showing they are close to a normal distribution.

(b) Plot of the residuals, showing no noticeable patterns.

Fig. 4. Descriptive statistics of the results of the linear regression model.

Qualitative Results. Since the participants did not just give an answer in the form of 'A is now enabled' but had to motivate their answers, some extra observations could be made concerning the results. Although it was the case that the two groups with the more elaborate tool were better capable of seeing which activities are enabled and which constraints are violated, they still seemed to ignore these annotations. Especially group B sometimes ignored the coloring of the model as they did not understand some implications of the constraints. Participants often also bended the descriptions of the Declare constraints towards their understanding, hence starting to discuss irrelevant parts of the model. For the third group, this behavior was still present, although to a much lesser extent. Group A participants often found the hidden dependencies in the easier examples. Because they had no support they analyzed the models thoroughly, but failed to find any hidden relations in the elaborate examples.

Remarks. As for all empirical studies, there are threats to validity that need to be addressed, the main ones in our case are:

- **Internal validity:** Our experiment had the maturation threat because subjects may react differently as time passes (because of boredom or fatigue). We solved this threat by dividing the experiment into different questions per model. Next this threat, we made sure there could be no interaction between the students of different sessions.
- **Construct validity:** Our experiment was threatened by the hypothesis guessing threat because students might figure out what the purpose of the study is, which could affect their guesses. We minimized this threat by hiding the goal of the experiment. Since the R-squared values were not very high, it might also be interesting to include the time spent on the questions and the grades of the final exam of the students to explain the score through the capabilities to learn and think logically in general.
- **External validity:** Our experiment might suffer from interaction of selection and treatment: the subject population is limited to students. Although the

number of subjects is quite high and their profiles balanced, we can only generalize the results to students. The subjects might not be representative to generalize the results to professional modelers as well. It is, e.g., not possible to claim that the tool can help or improve Declare modeling efforts of more experienced users.

6 Related Work

Declare, introduced as DecSerFlow and ConDec in [15,16], has become one of the most widely-used declarative process languages in research. Some competing approaches exist such as DCR Graphs [17], which are comparable to a slimmed-down version of Declare for improving understandability and setup, and the more data-oriented language Guard-Stage-Milestone [18].

Declare and its understandability has been researched for a test case-driven approach [4], the impact of hierarchies [1], and its common understandability challenges [2]. While these works clearly state the presence of hidden dependencies, with [2] explicitly mentioning this as a common pitfall for understandability, they have not provided a way to capture them. This work continues on the preliminary approach for retrieving hidden dependencies of [19].

Many other works on Declare mining exist as well, which have led to a better understanding of the properties of the language. Most notably the hierarchy [9] and semantics [10,20,21] and the transitivity properties [22] have brought clarification as to how constraints behave in a model.

7 Conclusion and Future Work

This paper shows how to retrieve and use dependency structures and unary propagation in Declare models to increase understandability. It offers a theoretic aspect in explaining how to construct and interpret the relations of constraints and their hidden dependencies in ways that have not been proposed yet, and was validated on novice users in an experiment. This showed that explaining and visualizing hidden dependencies and constraint structures rendered users significantly better capable of understanding the models.

Future work includes analyzing the results further by using extra user statistics such as average grades, as well as including new observations from expert users. Next, it is also straightforward to extend these findings to n-ary constraints, which changes only the propagation and interpretation slightly. Furthermore, constructing the hidden dependencies has numerous other applications. By understanding in which way constraints are related, it becomes easier to grasp the complexity of conjoining the separate Declare constraints' automata and hence it is possible to score the impact of different constraints on, e.g., the performance of calculating the global automaton of the whole model. Furthermore, these insights can be used to score a Declare model for simplicity., i.e., models which contain more hidden dependencies can be scored lower for this metric.

References

1. Zugal, S., Pinggera, J., Weber, B., Mendling, J., Reijers, H.A.: Assessing the impact of hierarchy on model understandability – a cognitive perspective. In: Kienzle, J. (ed.) MODELS 2011 Workshops. LNCS, vol. 7167, pp. 123–133. Springer, Heidelberg (2012)
2. Haisjackl, C., Zugal, S., Soffer, P., Hadar, I., Reichert, M., Pinggera, J., Weber, B.: Making sense of declarative process models: common strategies and typical pitfalls. In: Nurcan, S., Proper, H.A., Soffer, P., Krogstie, J., Schmidt, R., Halpin, T., Bider, I. (eds.) BPMDS 2013 and EMMSAD 2013. LNBIP, vol. 147, pp. 2–17. Springer, Heidelberg (2013)
3. Pesic, M., Schonenberg, H., van der Aalst, W.M.: Declare: full support for loosely-structured processes. In: 11th IEEE International Enterprise Distributed Object Computing Conference, EDOC 2007, pp. 287–300. IEEE (2007)
4. Zugal, S., Pinggera, J., Weber, B.: The impact of testcases on the maintainability of declarative process models. In: Halpin, T., Nurcan, S., Krogstie, J., Soffer, P., Proper, E., Schmidt, R., Bider, I. (eds.) BPMDS 2011 and EMMSAD 2011. LNBIP, vol. 81, pp. 163–177. Springer, Heidelberg (2011)
5. Montali, M., Pesic, M.M., Aalst, W., Chesani, F., Mello, P., Storari, S.: Declarative specification and verification of service choreographies. ACM Trans. Web **4**(1), 1–62 (2010)
6. De Smedt, J., De Weerdt, J., Vanthienen, J.: Fusion miner: process discovery for mixed-paradigm models. Decis. Support Syst. **77**, 123–136 (2015)
7. Burattin, A., Maggi, F.M., van der Aalst, W.M., Sperduti, A.: Techniques for a posteriori analysis of declarative processes. In: 2012 IEEE 16th International Enterprise Distributed Object Computing Conference, pp. 41–50. IEEE (2012)
8. Pesic, M.: Constraint-based workflow management systems: shifting control to users. Ph.D. thesis, Technische Universiteit Eindhoven (2008)
9. Di Ciccio, C., Mecella, M.: A two-step fast algorithm for the automated discovery of declarative workflows. In: 2013 IEEE Symposium on Computational Intelligence and Data Mining (CIDM), pp. 135–142. IEEE (2013)
10. Westergaard, M., Stahl, C., Reijers, H.A.: Unconstrainedminer: efficient discovery of generalized declarative process models. Technical report BPM-13-28, BPMcenter (2013)
11. Maggi, F.M., Montali, M., Westergaard, M., van der Aalst, W.M.P.: Monitoring business constraints with linear temporal logic: an approach based on colored automata. In: Rinderle-Ma, S., Toumani, F., Wolf, K. (eds.) BPM 2011. LNCS, vol. 6896, pp. 132–147. Springer, Heidelberg (2011)
12. De Smedt, J., De Weerdt, J., Vanthienen, J., Poels, G.: Mixed-paradigm process modeling with intertwined state spaces. Bus. Inf. Syst. Eng. **58**(1), 19–29 (2016)
13. Westergaard, M., Maggi, F.M.: Declare: a tool suite for declarative workflow modeling and enactment. BPM (Demos) **820**, 1–5 (2011)
14. Møller, A.: dk. brics. automaton-finite-state automata and regular expressions for Java (2010). Accessed 30 Aug 2014
15. van der Aalst, W.M.P., Pesic, M.: DecSerFlow: towards a truly declarative service flow language. In: Bravetti, M., Núñez, M., Zavattaro, G. (eds.) WS-FM 2006. LNCS, vol. 4184, pp. 1–23. Springer, Heidelberg (2006)
16. Pesic, M., van der Aalst, W.M.P.: A declarative approach for flexible business processes management. In: Eder, J., Dustdar, S. (eds.) BPM Workshops 2006. LNCS, vol. 4103, pp. 169–180. Springer, Heidelberg (2006)

17. Hildebrandt, T.T., Mukkamala, R.R.: Declarative event-based workflow as distributed dynamic condition response graphs. In: PLACES, pp. 59–73 (2011)
18. Hull, R., et al.: Introducing the guard-stage-milestone approach for specifying business entity lifecycles (invited talk). In: Bravetti, M. (ed.) WS-FM 2010. LNCS, vol. 6551, pp. 1–24. Springer, Heidelberg (2011)
19. De Smedt, J., De Weerdt, J., Serral Asensio, E., Vanthienen, J.: Gamification of declarative process models for learning and model verification. In: Business Process Management Workshops. Springer (2015). Accepted
20. 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, pp. 1027–1033, 27–31 July 2014, Québec City, Québec, Canada (2014)
21. Di Ciccio, C., Mecella, M., Mendling, J.: The effect of noise on mined declarative constraints. In: Ceravolo, P., Accorsi, R., Cudre-Mauroux, P. (eds.) Data-Driven Process Discovery and Analysis, pp. 1–24. Springer, Heidelberg (2015)
22. Maggi, F.M., Bose, R.P.J.C., van der Aalst, W.M.P.: A knowledge-based integrated approach for discovering and repairing declare maps. In: Salinesi, C., Norrie, M.C., Pastor, Ó. (eds.) CAiSE 2013. LNCS, vol. 7908, pp. 433–448. Springer, Heidelberg (2013)