Configurable Declare: Designing Customisable Flexible Process Models[*,**]

Dennis M.M. Schunselaar, Fabrizio Maria Maggi,
Natalia Sidorova, and Wil M.P. van der Aalst

Department of Mathematics and Computer Science,
Eindhoven University of Technology,
P.O. Box 513, 5600 MB, Eindhoven, The Netherlands
{d.m.m.schunselaar,f.m.maggi,n.sidorova,w.m.p.v.d.aalst}@tue.nl

Abstract. Declarative languages are becoming more popular for modelling business processes with a high degree of variability. Unlike procedural languages, where the models define what is to be done, a declarative model specifies what behaviour is not allowed, using constraints on process events. In this paper, we study how to support configurability in such a declarative setting. We take *Declare* as an example of a declarative process modelling language and introduce *Configurable Declare*. Configurability is achieved by using configuration options for event hiding and constraint omission. We illustrate our approach using a case study, based on process models of ten Dutch municipalities. A Configurable *Declare* model is constructed supporting the variations within these municipalities.

Keywords: business process modelling, configurable process models, declarative process models, Declare.

1   Introduction

Process-aware information systems [4], such as workflow management systems, case-handling systems and enterprise information systems, are used in many branches of industry and governmental organisations. *Process models* form the heart of such systems since they define the flow of task executions. Traditionally, the languages used for process modelling are *procedural* languages, since they are very appropriate for describing well-structured processes with a predefined flow. At the same time, procedural models become very complex for environments with high variability, since every possible execution path has to be encoded in the model. In the most extreme cases, like specifications of some medical
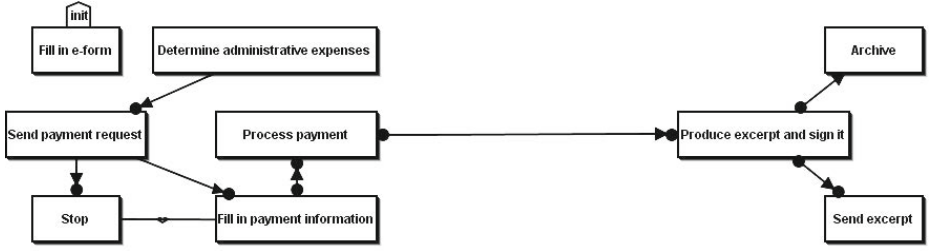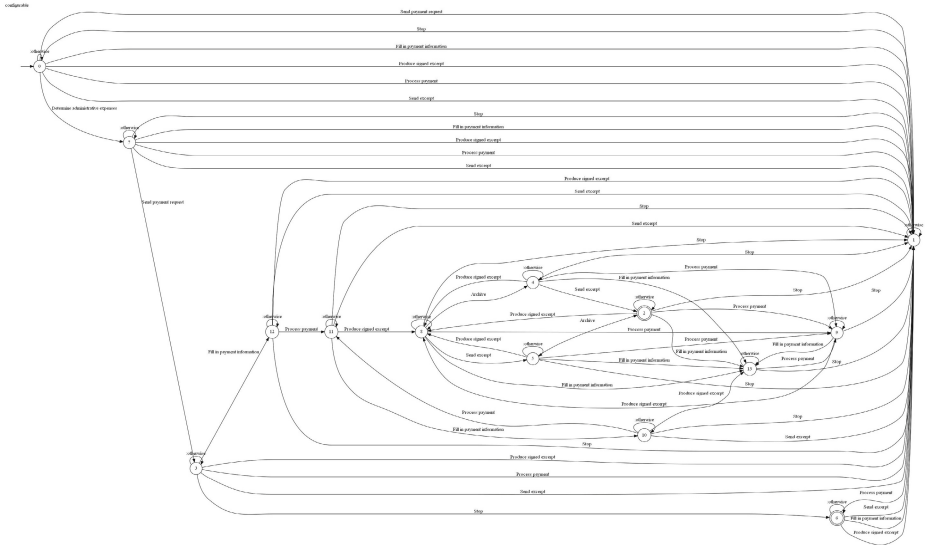
Fig. 1. Example of Declare model describing the process for requesting an excerpt from the civil registration (Mun. A)

Fig. 2. Automaton obtained from the translation of the Declare model in Fig. 1

protocols, the process flow cannot be completely predefined, and the procedural way of modelling becomes impossible.

Unlike procedural languages, specifying what should be done, *declarative* languages specify which constraints may not be violated, and therefore they allow for comprehensible descriptions of processes with a high degree of variability. Declarative languages are also very appropriate for defining *compliance models*, which specify *what* should (not) be done instead of saying how it should be done. Consider, for instance, the set of rules in Fig. 1 from our case study. By translating this simple set of rules into an automaton[1], we obtain the procedural model in Fig. 2 that is much more complex.

---

[1] We have used the Declare tool (`http://www.win.tue.nl/declare/`) for the translation of Declare models to automata.

In recent years, a number of declarative process modelling languages were developed [1,10,11,14] and proven to be more suitable for certain application domains than procedural languages [6,12,19]. Nevertheless, since declarative process modelling languages attracted the attention of the research community at the later stage, when procedural languages were already massively used, there are still serious gaps in the domain of declarative process modelling which are still to be filled in. In this paper, we address one such a gap: *configurability of declarative process models*.

Nowadays, many branches of industry have semi-standardised collections of process models. Within one branch, process models of different organisations are often very similar due to legislation and (partial) standardisation, e.g., processes for registering a birth or extending a driving license would be very similar to each other for different municipalities. A one-size-fits-all approach with full standardisation of processes is however often inappropriate, as these organisations have good reasons for using specific variants of these common processes.

Configurable process models were introduced to solve the aforementioned problems [8,9]. They allow the user to change some parts of the model towards the user's preferences. This solves the one-size-fits-all problem and improves maintainability of processes since it becomes possible to describe several slightly different models by a single configurable model. When the configurable model is changed, all process models are updated automatically. To the best of our knowledge, the only kind of existing configurable process models are *procedural* models.

In this paper, we study in which way configurability in the declarative context is different from configurability in the procedural context. For this purpose, we consider the example declarative language *Declare* [11], in which constraints are LTL-formulas evaluated on traces of events executed in a process, and we define *Configurable Declare*. Since a declarative process model is a set of constraints over a set of events (representing the completion of a specific task in a business process), the *configuration options* we include in *Configurable Declare* are (1) *hiding an event* and (2) *omitting a constraint*. Through a *configuration*, it is possible to specify, for each configuration option, a boolean value that indicates whether a hideable event must be hidden or an omittable constraint must be omitted in the configured model.

Like most declarative languages, *Declare* works under the open world assumption, and, therefore, hiding an event does not mean forbidding this event to be executed. As the name suggests, hiding means allowing some event to become unobservable, unmonitored, unlogged. The behaviour of a model in which an event is hidden should remain the same as it was modulo this event. In the case of a *Declare* model, where process behaviour is considered to be defined by the set of traces (language) compliant with the model, hiding an event should result in a model with the same language modulo the hidden event.

To achieve language preservation, we take into account implicit constraints that would be lost if we simply removed from the model the event and the constraints connected to the event. For example, consider a model with constraints

"every paper submission is followed by a review" and "every review is followed by sending a notification letter" in which the event "review" gets hidden. This implies that the two constraints we have will be removed together with the event. To preserve the language modulo the hidden event "review", we have to include the implicit constraint "every paper submission is followed by sending a notification letter" into the configured model. We define a derivation scheme for implicit constraints that allows us to have a sound transformation of a configurable model and a configuration to a configured model.

Since some configurations might lead to uninteresting or undesirable process variants, we introduce meta-constraints, which are defined as logical expressions over configuration choices, e.g., "if event A is hidden, then event B is not hidden". Meta-constraints, in fact, restrict possible configurations to configurations that make sense from the content-wise perspective.

To support process modelling with *Configurable Declare*, we have developed **ConfDeclare** [16]. We have used this tool in the context of the *CoSeLoG*[2] project for a case study based on process models from ten Dutch municipalities. These models represent the production of an excerpt from the civil registration. In particular, starting from the different variants of the process (one for each municipality) we build a *Configurable Declare* model from which these variants can be derived.

Related Work. Configurable process models have been defined for a number of procedural modelling languages, e.g., *C-SAP WebFlow*, *C-BPEL*, *C-YAWL* [8], *CoSeNets* [18], and *C-EPC* [13]. Imperative configurable process models support a number of standard operations. Some patterns for procedural configurable models have been identified to support these operations [2,3,8,15]. *Configurable Declare* supports those patterns that can be mirrored to the declarative approach.

The paper is structured as follows: Section 2 gives a brief introduction to *Declare*. In Section 3, we introduce *Configurable Declare*. Section 4 explains the configuration steps: hiding an event and omitting a constraint. In Section 5, we show how to derive a *Declare* model from a *Configurable Declare* model and a configuration. Finally, in Section 6, we draw some conclusions and discuss directions for future work.

2   Declare: A User-Friendly Declarative Language

A process can be described by using different types of modelling languages. Process modelling languages can be classified according to two categories: procedural and declarative. A *procedural* model works with a "closed world" assumption, i.e., it explicitly specifies all the acceptable sequences of events in the process and everything that is not mentioned in the model is forbidden. Procedural process models can be used to provide a high level of operating support to participants

---

[2] `http://www.win.tue.nl/coselog/`

Table 1. FLTL operators semantics

operator	semantics
$\bigcirc\varphi$	φ has to hold in the next position of a path.
$\square\varphi$	φ has to hold in all the subsequent positions of a path.
$\Diamond\varphi$	φ has to hold eventually (somewhere) in the subsequent positions of a path.
$\varphi\,U\psi$	φ has to hold in a path at least until ψ holds and ψ must hold in the current or in some future position.

who simply follow one of the allowed sequences in the model during the process execution. Therefore, this type of models is optimal in environments that are stable and where the process flow can be fully described in the model.

In contrast, a *declarative* model describes a process through constraints that should not be violated by the process execution. A declarative process model works with an "open world" assumption, i.e., any event is allowed unless it is explicitly forbidden by some constraint. This type of models can be used in highly dynamic environments where processes have a low degree of predictability. This is optimal when participants make decisions themselves and adapt the process flow accordingly (e.g., a doctor in a procedure to treat a fracture). Using declarative models for such processes allows for compact, readable representations.

In this paper, we study configurability of declarative process modelling languages taking *Declare* as an example of these languages. We explain here the basics of *Declare* necessary in the context of configurability and we refer the reader to [11] for a complete description of the language. *Declare* has formal semantics based on the use of a temporal logic, but the modeler is not confronted with this formal side, since the language has a user-friendly graphical notation capturing behavioural patterns expressible as temporal logic formulas. These patterns are a superset of the ones identified by Dwyer et al. in [5] and each of them has a specific graphical notation and semantics.

Given that business processes eventually terminate, *Declare* reasons on finite traces of events and uses a variant of LTL for finite traces called FLTL [7]. Table 1 contains the main FLTL operators and their semantics. Fig. 3 shows the graphical notation for the *response* constraint ($response(A, B)$) in *Declare*. The semantics of this constraint is captured in FLTL by $\square(A \Rightarrow \Diamond B)$ ("every occurrence of event A is eventually followed by an occurrence of event B). In Table 2, we summarise the graphical notation and the FLTL semantics of the *Declare* constraints used in this paper.

A *Declare* model can be seen as a set of constraints, i.e., a conjunction of FLTL formulas over events. Formally, a *Declare* model can be defined as follows:

Fig. 3. The response constraint between A and B

Table 2. Graphical notation and FLTL semantics of the *Declare* constraints used in this paper

constraint	FLTL semantics	graphical notation
$respone(A, B)$	$\Box(A \Rightarrow \Diamond B)$	
$precedence(A, B)$	$(\neg B \ U A) \vee \Box(\neg B)$	
$succession(A, B)$	$response(A, B) \wedge precedence(A, B)$	
$alternate\ response(A, B)$	$\Box(A \Rightarrow \bigcirc(\neg A \ U B))$	
$exclusive\ 1\ of\ 2(A, B)$	$(\Diamond A \wedge \neg \Diamond B) \vee (\neg \Diamond A \wedge \Diamond B)$	
$init(A)$	A	

Definition 1 (Declare Model). *A **Declare model** is a pair M=(E, C), where E is a set of events and C is a set of FLTL formulas over events in E.*

A trace of events belongs to the language of a *Declare* model, if it satisfies all the constraints of this model:

Definition 2 (Language). *The* language *of a Declare model $M = (E, C)$, namely $\mathcal{L}(M)$, is the set of all traces satisfying all the constraints from C.*

The formal semantics allows every *Declare* model to be executable and verifiable. To verify the validity of a constraint on a trace, the corresponding FLTL formula can be translated to a finite state automaton that accepts those and only those traces on which the formula is satisfied. The automaton for the response constraint in Fig. 3 is shown in Fig. 4; the initial state (marked by an edge with no origin) is here also an accepting state (indicated using a double outline). When event A happens, the state of the automaton is changed to a non-accepting state; it changes back to the initial accepting state only when a event B happens. Next to the positive labels, we also have negative labels (e.g., $\neg A$). They indicate that we can follow the transition for any event not mentioned (e.g., we can execute event C from the initial state and remain in the
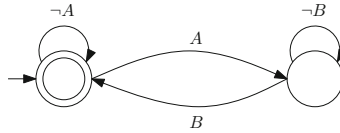
Fig. 4. Automaton for $response(A, B)$

same state). This allows us to use the same automaton regardless of the input language, taking into account the open-world assumption. To verify the validity of a *Declare* model $M = (E, C)$ for a trace, we consider the automaton obtained from the conjunction of the constraints in the model.

Fig. 1 shows a simple *Declare* model describing the process for requesting an excerpt from the civil registration in a Dutch municipality. This model is part of a case study conducted in collaboration with ten Dutch municipalities in the *CoSeLoG* project. These municipalities model (and execute) this process in different, but still very similar ways, which makes it interesting in the context of configurability.

The *Declare* model in Fig. 1 involves nine *events*, depicted as rectangles, (e.g., *Send payment request*) and nine *constraints*, showed as connectors between the events (e.g., *succession*). Events represent the completion of a specific task in the business process. Constraints highlight mandatory and forbidden behaviours, implicitly identifying the acceptable sequences of events that comply with the model.

The constraint *init* shows that *Fill in e-form* must be the first task performed in the process; note that it is not forbidden to execute this task again later on. Since it is necessary to exactly evaluate the total amount of administrative expenses before formulating a payment request, *Send payment request* can be performed only after having performed *Determine administrative expenses*, as indicated by the *precedence* constraint between the two events. *Fill in payment information* must eventually be followed by *Process payment*, and *Process payment* cannot complete before that *Fill in payment information* is completed, which is captured by the *succession* constraint, being the combination of *precedence* and *response* constraints. The *exclusive choice* between events *Stop* (that stops the procedure) and *Fill in payment information*, depicted as a line with a black diamond, indicates that one of these two events must be performed in the process but not both. The *response* constraint between *Produce excerpt and sign it* and *Archive* indicates that whenever *Produce excerpt and sign it* is executed, *Archive* must eventually follow.

3   Configurable Declare

In this section, we introduce *Configurable Declare*, an extension of *Declare*. While in imperative languages, where the focus is on modelling the allowed behaviour, configuration options aim at making some behaviour optional (e.g., by blocking an event), in declarative languages the focus is on modelling restrictions on the behaviour, and therefore our configuration options will aim at making the restrictions optional. One reason to make some restrictions in the model optional can come from the inability to execute, control or monitor an event in some context. In this case, we want to allow for hiding an event depending on the configuration chosen. Another way to remove some restriction is by removing constraints from the model. Therefore, *Configurable Declare* introduces the possibility of annotating some constraints as omittable. Finally, we use meta-constraints to define which options for configuring the model are combinable and which not.

(a) A hideable event   (b) An omittable constraint
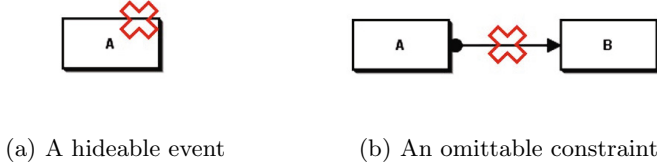
Fig. 5. Graphical representation of the configuration options

Formally, we define a *Configurable Declare* model as:

Definition 3 (Configurable Declare Model). *A Configurable Declare* model *is a tuple* (M, E_h, C_o, MC) *where:*

- $M = (E, C)$ *is a Declare model,*
- $E_h \subseteq E$ *is a set of* hideable events *(graphically represented as in Fig. 5(a)),*
- $C_o \subseteq C$ *is a set of* omittable constraints *(graphically represented as in Fig. 5(b)), and*
- *MC is a set of* meta-constraints *and each meta-constraint defines a narrowing of the function space* $(E_h \cup C_o) \to \mathbb{B}$. *We call this narrowing the* configuration space *of the Configurable Declare model.*

For the sake of readability, we overload the notation used and write, for example, $\neg e \Rightarrow (\neg c_1 \land \neg c_2)$ for the meta-constraint saying that if hideable event e is hidden, then omittable constraints c_1 and c_2 must be omitted, which implies that the configuration space only include functions, whose values on e, c_1 and c_2 obey the meta-constraint.

To configure a *Configurable Declare* model, the user has to make, for each hidable event and omittable constraint, a configuration choice specifying whether a hideable event should be hidden, and whether an omittable constraint should be omitted in the configured model.

Definition 4 (Configuration). *Let* $M_{conf} = (M, E_h, C_o, MC)$ *be a Configurable Declare model. A* configuration *of* M_{conf} *is a function conf* $: (E_h \cup C_o) \to \mathbb{B}$ *from the configuration space of* M_{conf}.

By applying a configuration to a *Configurable Declare* model, we obtain a configured model, which is one of the possible variants deducible from the given *Configurable Declare* model.

4 Configuration Steps

We choose a two-step approach for defining a configuration of a *Configurable Declare* model. In the first step *–abstraction–* the user defines the part of the configuration that specifies which hideable events must be hidden in the configured model, thus setting the context for this model. The *Configurable Declare*
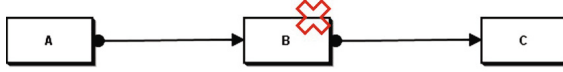
Fig. 6. An implicit constraint between A and C can disappear when removing B

model is, then, transformed into a modified *Configurable Declare* model obtained by hiding the events that must be hidden according to the configuration. In the second step –*configuring constraints*– the user defines the part of the configuration that specifies which omittable constraints must be omitted in the configured model. Starting from the modified *Configurable Declare* model obtained in the first step, the configured model is derived by omitting the constraints that must be omitted according to the defined configuration.

4.1   Abstraction

In the abstraction step, the user defines a configuration over the hideable events by choosing (not) to hide events that are hideable in the *Configurable Declare* model. Note that hiding an event in a *Declare* model does not mean that it cannot occur, but it means that it is not monitored, implying that there can be no constraints restricting the execution of that event.

In this step, the user can possibly hide events which are involved in *implicit constraints*, i.e., constraints that can be deduced from other (explicit) constraints. Consider, for instance, the model in Fig. 6 where every A is eventually followed by B, and every B is eventually followed by C. By transitivity, also every A is eventually followed by C, which is an implicit constraint. Similarly to hiding in other settings (e.g., in the process algebra context), we want a model derived by hiding B to be *visible-language-equivalent* to the model where B is not hidden (with the hidden event considered to be invisible, like a τ-event). When hiding B, we cannot simply remove B together with all the constraints connected to B, since this would also remove the implicit constraint between A and C. To maintain the language equivalence, we have to take the implicit constraints into account, and, when necessary, make implicit constraints explicit.

Table 3 presents an excerpt from the list of constraint combinations connecting events A, B and C, and the corresponding implicit constraints that should be

Table 3. An excerpt of the language equivalences after the τ-abstraction of B

$$
\begin{aligned}
\mathcal{L}^{B\leftarrow\tau}(response(A,B) \wedge response(B,C)) &= \mathcal{L}(response(A,C)) \\
\mathcal{L}^{B\leftarrow\tau}(response(A,B) \wedge succession(B,C)) &= \mathcal{L}(response(A,C)) \\
\mathcal{L}^{B\leftarrow\tau}(precedence(A,B) \wedge precedence(B,C)) &= \mathcal{L}(precedence(A,C)) \\
\mathcal{L}^{B\leftarrow\tau}(precedence(A,B) \wedge succession(B,C)) &= \mathcal{L}(precedence(A,C)) \\
\mathcal{L}^{B\leftarrow\tau}(succession(A,B) \wedge response(B,C)) &= \mathcal{L}(response(A,C)) \\
\mathcal{L}^{B\leftarrow\tau}(succession(A,B) \wedge precedence(B,C)) &= \mathcal{L}(precedence(A,C)) \\
\mathcal{L}^{B\leftarrow\tau}(succession(A,B) \wedge succession(B,C)) &= \mathcal{L}(succession(A,C))
\end{aligned}
$$
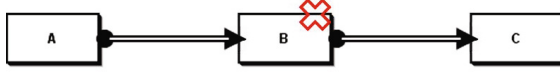
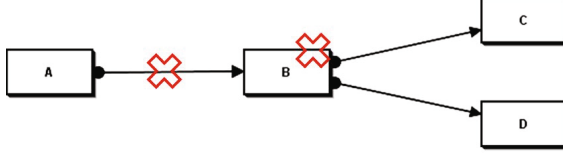**Fig. 7.** The implicit constraint between A and C is not expressible in standard *Declare*

added when hiding B. In [16], the reader can find the full list of combinations of standard *Declare* constraints with the strongest implicit constraint (expressable in the standard *Declare*) corresponding to each combination. Several of these combinations do not allow for maintaining the language equivalence and the best we can do is to approximate the implicit constraint. Consider, for instance, the model in Fig. 7 consisting of two *alternate response* constraints. When B is not hidden, the configured model accepts (among others) traces $ABCABC$ and $ABACBC$, which are abstracted to $ACAC$ and $AACC$ when hiding B in the traces. Therefore, when B is hidden, the configured model does not accept any trace in which more than two occurrences of A happen without a C in between. By removing B from the model (i.e., by substituting B by τ), we would need to obtain a *Declare* model which accepts exactly such traces, but such a constraint is not expressible in *Declare*.

The approach provided in [16] considers as implicit constraint the closest constraint that is stronger than the necessary (inexpressible) constraint, which is *alternate response*(A, C) for the example considered (this constraint does not accept the trace $AACC$.) An alternative approach is to choose for the closest weaker constraint. This issue can be fully overcome by extending *Declare* with new constraints (in this case with a constraint that forces the occurrence of C after two occurrences of A). When using only the standard *Declare* constraints, the user should be notified in case the language preservation is violated.

Below we sketch the proof idea, supporting our method for hiding events. The proof is done by comparing the languages of the models obtained from the same *Configurable Declare* model using different configurations.

Theorem 1. *Let $M = (E, C)$ and $M' = (E', C')$ be Declare models obtained by not hiding/hiding an event $e \in E$ in the Configurable Declare model $((E, C), \{e\}, \emptyset, \emptyset)$, respectively. Then, $\mathcal{L}^{e \leftarrow \tau}(M) = \mathcal{L}(M')$, i.e., the language of model M with event e considered as invisible is the same as the language of model M'.*

Proof. (Idea) We can transform M to an equivalent model M'' where the implicit constraints in which e is involved are made explicit, maintaining the language equivalence between M and M''. Considering that implicit constraints are deducible from the explicit constraints in the model, this implies that we do not constrain the behaviour any further. Afterwards, we transform M'' to M' by removing event e and all constraints associated with e. Showing that $\mathcal{L}^{e \leftarrow \tau}(M'') = \mathcal{L}^{e \leftarrow \tau}(M')$ is straightforward, using the equivalences as shown in Table 3. Therefore, we can conclude that model M is visible-language equivalent to the model M' (with e abstracted to τ). □

Fig. 8. Implicit constraints between A and C and between A and D are omittable

When a hidden event is involved in omittable constraints, the situation becomes slightly more complicated. If an implicit constraint to be added to the model is derived using an omittable constraint, the implicit constraint should be added as an omittable constraint. Consider, for instance, the model in Fig. 8. The constraint between A and B can be omitted and B can be hidden. If we choose to hide B, we have to make explicit the implicit constraints between A and C (c_1) and between A and D (c_2). However, since the constraint between A and B is omittable, we have to make the implicit constraints c_1 and c_2 also omittable.

At the same time, we need to preserve correlations between implicit constraints. Consider, again, the model in Fig. 8. If the constraint between A and B is omitted, then both the implicit constraints c_1 and c_2 disappear. This introduces a correlation between the implicit constraints c_1 and c_2: they should be either both present or both omitted in the configured model. This can be encoded through meta-constraints as $c_1 \Leftrightarrow c_2$.

Deducing which correlations have to be maintained between pairs of implicit constraints is straightforward. Consider, for instance, the models in Fig. 9, in which all implicit constraints have been made explicit. Constraint c_4 is deduced from c_1 and c_3, and c_5 is deduced from c_2 and c_3. In the model in Fig. 9(a), explicit constraints c_1 and c_2, and c_3 are all omittable. Starting from the omittable (explicit) constraints, we can build the deduction graph in Fig. 10(a). In this graph, if explicit constraints are used to deduce an implicit constraint, we include an hyperarc between the explicit constraints and the implicit constraint. For instance, c_4 is deduced from c_1 and c_3, hence, we include a hyperarc between c_1, c_3 and c_4. Using the hyperarcs, we can induce meta-constraints like $(c_1 \wedge c_3) \Leftrightarrow c_4$.

(a) model 1   (b) model 2
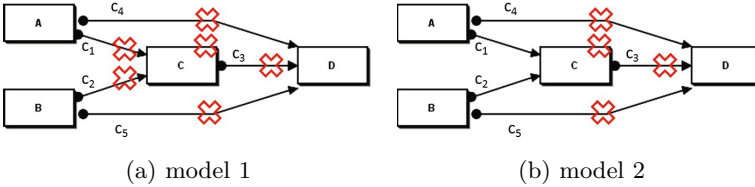
Fig. 9. *Configurable Declare* models with the implicit constraints (c_4, c_5) made explicit

(a) Deduction graph of model 1   (b) Deduction graph of model 2
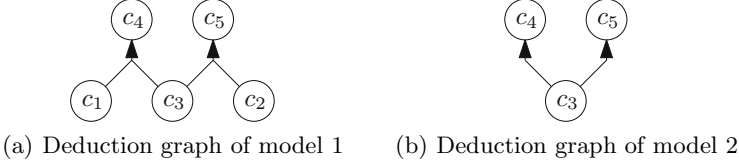
Fig. 10. Deduction graphs of the *Configurable Declare* models in Fig. 9
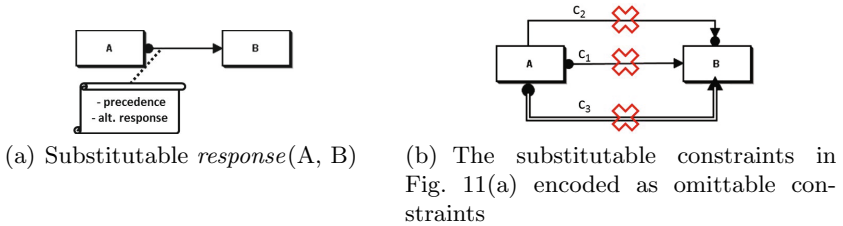
Using the deduction graphs, we can easily induce correlations between implicit constraints. Consider the *Configurable Declare* model in Fig. 9(b) and the corresponding deduction graph in Fig. 10(b). From Fig. 10(b), we can induce the meta-constraints $c_3 \Leftrightarrow c_4$ and $c_3 \Leftrightarrow c_5$. By transitivity, we obtain the meta-constraint $c_4 \Leftrightarrow c_5$. Therefore, when hiding C, c_4 and c_5 must be both omitted or both not omitted.

From the technical perspective, it would be easier to let the user first define which constraints should be omitted and then choose which events should be hidden. In this case, omittability of implicit constraints and correlations between them would be irrelevant. This would, however, require the user to make choices about constraints that are defined on events which are not relevant for her practice.

4.2 Configuring Constraints

In the second step, the user can decide which omittable constraints should be omitted in the configured model. It is also possible to include the option of substituting a constraint by a different constraint into *Configurable Declare*. However, this option can be considered as syntactic sugar. Indeed, substituting a (default) constraint with different constraints can be obtained by considering these constraints as omittable and providing a meta-constraint specifying that exactly one of this constraints can be kept in the configured model.

In Fig. 11(a), for instance, we show a model in which the *response*(A, B) can be substituted by either *precedence*(A, B) or by *alternate response*(A, B). This can also be encoded through omittable constraints as depicted in Fig. 11(b) with a meta-constraint enforcing that exactly one of the omittable constraints is kept in the configured model. This is encoded as $c_1 \veebar c_2 \veebar c_3$, where $\veebar$ is the exclusive or.

(a) Substitutable *response*(A, B)   (b) The substitutable constraints in Fig. 11(a) encoded as omittable constraints

Fig. 11. Substitutable constraints in *Configurable Declare*

5 Methodology and Case Study

In this section, we present the deduction of a *Declare* model from a *Configurable Declare* model. As mentioned before, this is done through a two-step approach. First the context is set (Subsection 5.1), and then, some constraints are selected to be omitted in the configured model (Subsection 5.2). The methodology is presented by using an example from our case study. Further results about the case study are given in Section 5.3.

5.1 Setting the Context

The first step for configuring a *Configurable Declare* model consists in selecting which events should be controlled and which events are uncontrolled in the configured model. Using Algorithm 1, the *Configurable Declare* model is transformed into a (modified) *Configurable Declare* model. Here, the hideable events that are chosen to be hidden (denoted as set S_h) are removed from the model. It can be the case that hiding an event invalidates some meta-constraints, or even that the event to be hidden is not hideable. Therefore, we first check whether all meta-constraints are satisfied and whether all events in S_h are hideable ($S_h \subseteq E_h$). If this is the case, the events in S_h are removed from the *Configurable Declare*

Algorithm 1: Setting the context for a *Configurable Declare* model
SETTHECONTEXT(M_{conf}, S_h, $M_{conf'}$)
Input: M_{conf} the *Configurable Declare* model, $S_h \subseteq E_h$ the set of hidden events
Output: $M_{conf'}$ the *Configurable Declare* model after abstraction
(1)   **if** the meta-constraints MC are not satisfied
(2)   **return**
(3)   **else**
(4) $M_{temp} \leftarrow M_{conf}$
(5) $\mathcal{T} \leftarrow$ all implicit constraints (using the transitive closure based on the rules in [16])
(6)   **foreach** event $e \in M_{temp}$
(7)   **if** $e \in S_h$
(8) $\mathcal{C} \leftarrow$ all implicit constraints which have to be made explicit after hiding e using the defined rules (see [16])
(9)   add all constraints from $\mathcal{C}$ to M_{temp}
(10)   add all meta-constraints related to the removal of e (based on $\mathcal{T}$) to M_{temp}
(11)   Remove all events from M_{temp} which are hidden
(12)   Remove all constraints from M_{temp} which are related to hidden events
(13)   Update all meta-constraints in M_{temp} which are related to hidden events
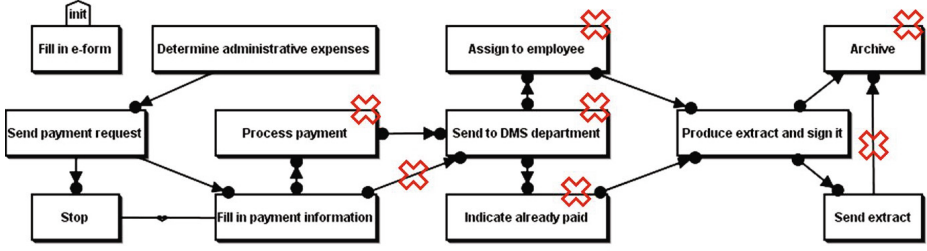(14)   **return** M_{temp}

Fig. 12. The *Configurable Declare* model

model, the implicit constraints are made explicit if needed, and meta-constraints are updated accordingly. Otherwise, the empty model is returned.

Consider the *Configurable Declare* model in Fig. 12 (without any specified meta-constraint at the beginning). Suppose that the user chooses to hide events *Assign to employee*, *Send to DMS department*, and *Indicate already paid*.

If we hide *Send to DMS department*, we need to add to the modified model ($M_{conf'}$) a *succession* constraint between *Process payment* and *Assign to employee* (c_1), and between *Process payment* and *Indicate already paid* (c_2). Furthermore, we have to include in $M_{conf'}$ a *succession* constraint between *Fill in payment information* and *Assign to employee* (c_3), and between *Fill in payment information* and *Indicate already paid* (c_4). Since the *succession* between *Fill in payment information* and *Send to DMS department* can be omitted, we have to maintain the correlation between c_3 and c_4, i.e., the meta-constraint $m_1 = c_3 \Leftrightarrow c_4$ is added to $M_{conf'}$.

In the second iteration, we process *Assign to employee*. This introduces a *succession* between *Process payment* and *Produce extract and sign* (c_5) in $M_{conf'}$. Furthermore, the *succession* between *Fill in payment information* and *Produce extract and sign it* (c_6) has to be made explicit, and we have to include the meta-constraint $m_2 = c_4 \Leftrightarrow c_6$ in $M_{conf'}$.

If we hide *Indicate already paid*, we need to add to $M_{conf'}$ the constraints *succession*(*Process payment*, *Produce extract and sign it*) and *succession*(*Fill in payment information*, *Produce extract and sign it*). Note that no new meta-constraints have to be included at this iteration. Finally, we have to remove the hidden events from the model, and remove the constraints and update the meta-constraints related to any of those events. This yields the *Configurable Declare* model depicted in Fig. 13 (with meta-constraints m_1 and m_2).

5.2   Configuring Constraints

The second step for configuring a *Configurable Declare* model consists of selecting which constraints have to be omitted in the configured model (we indicate this set of constraints with S_o). Omitting a constraint might invalidate some meta-constraints, or it can be the case that the constraint to be omitted is not omittable. Therefore, we first check whether all meta-constraints are satisfied
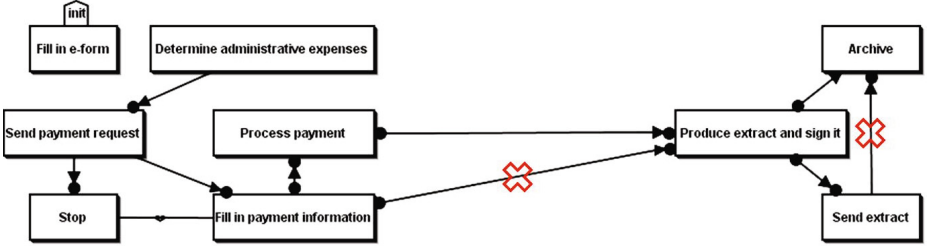
Fig. 13. The *Configurable Declare* model for municipality A after setting the context

and whether all constraints in S_o are omittable ($S_o \subseteq C_o$). If this is the case, the constraints in S_o are removed from the *Configurable Declare* model. Otherwise, the empty model is returned (Algorithm 2).

Suppose that we start from the *Configurable Declare* model depicted in Fig. 13 (with meta-constraints m_1 and m_2). Suppose that the user chooses to remove the *succession* between *Fill in payment information* and *Produce extract and sign it*, and the *precedence* between *Archive* and *Send excerpt*. Since m_1 and m_2 are satisfied, *succession* and *precedence* can be removed from the model yielding the model depicted in Fig. 1, which belongs to municipality A.

5.3   Case Study

For the case study, we have used models from the *CoSeLoG* project adopted by ten different Dutch municipalities. We have used the model for municipality A (depicted in Fig. 1) as running example to present our proposed approach. To obtain the models depicted in Fig. 14 and in Fig. 15 for municipalities B and C, we use the configurations showed in Table 4 and in Table 5.

Algorithm 2: Removing omitted constraints from the *Configurable Declare* model

CONFIGURABILITY$(((E, C), E_h, C_o, MC), S_o, M_{conf'})$

Input: $((E, C), E_h, C_o, MC)$ the *Configurable Declare* model, $S_o \subseteq C_o$ the set of omitted constraints

Output: M_{decl} the *Declare* model after omitting the constraints

(1)   **if** the meta-constraints MC are not satisfied

(2)   **return**

(3)   **else**
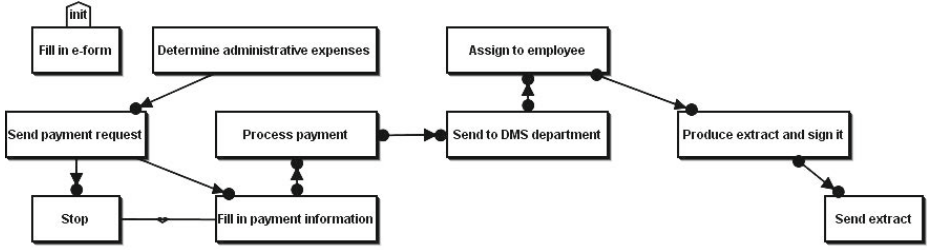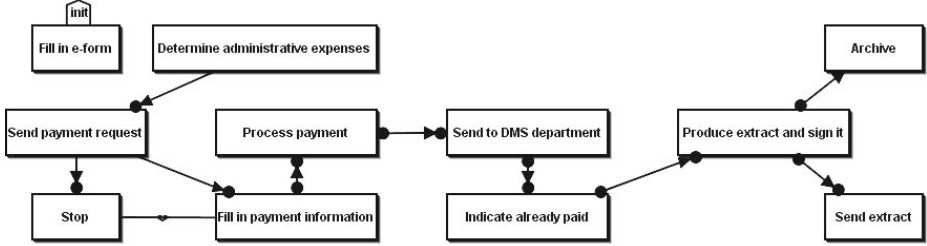
(4)   **return** $(E, C \setminus S_o)$

Fig. 14. The *Declare* model for municipality B

Fig. 15. The *Declare* model for municipality C

Table 4. The context for municipalities B and C

Event	Mun. B	Mun. C
Archive	hidden	not hidden
Assign to employee	not hidden	hidden
Indicate already paid	hidden	not hidden
Process payment	not hidden	hidden
Send to DMS department	not hidden	not hidden

Table 5. The configurability for municipalities B and C

Constraint	Mun. B	Mun. C
precedence(*Archive*, *Send Extract*)	omitted	omitted
succession(*Fill in payment information*, *Send to DMS department*)	omitted	omitted

6   Conclusion

In this paper, we defined *Configurable Declare*, a configurable declarative language. The configurability setting for declarative languages differs from the setting for procedural languages. Indeed, while adding configurability options for procedural languages implies that *more options for allowed behaviour* get included in the model, adding configurability options for declarative languages results in the inclusion of *more options for restricting behaviour*.

We have defined an approach to transform a *Configurable Declare* model and a given configuration into a *Declare* model. While in the declarative setting removing a constraint turned out to be a trivial transformation, in the procedural

setting removing a dependency between two events without influencing dependencies between other events is far from being trivial. On the other hand, hiding an event is easy to implement in the procedural setting, whereas it requires a dedicated mechanism to maintain implicit constraints in the declarative setting.

We have applied our approach as a proof of concept to a case study and we have been able to capture processes of ten Dutch municipalities in one readable *Configurable Declare* model. This paper must be considered as a starting point for *Configurable Declare* and there are several research directions we want to investigate concerning this topic. Below we elaborate on some of them.

Outlook. Building a *Configurable Declare* model from scratch is a logical option when a completely new process needs to be designed. However, in many cases (like in our case study) organisations already have models of their processes available and the configurable model should be built based on some existing knowledge. To make it possible, we are working on an approach for automatic generation of a *Configurable Declare* model from a given set of *Declare* models in such a way that the original *Declare* models are derivable from the generated *Configurable Declare* model (by applying some configurations). A related question is how to derive automatically a configuration for a given *Configurable Declare* model resulting in a model that is similar to a given *Declare* model.

When an organisation wants to start configuring a configurable model for some existing process for which no model is available, event logs can be used for deriving an appropriate configuration.

Finally, we would like to introduce patterns for meta-constraints in order to ease the design process. In particular, we want to develop a method for the automated deduction of meta-constraints to forbid configurations that lead to unsatisfiable models (models with no behaviour), or to models in which some important events become not executable or some important constraints become trivially true [17].

References

1. 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. Internationales Begegnungs- und Forschungszentrum fuer Informatik (IBFI), Schloss Dagstuhl, Germany (2006)
2. Becker, J., Delfmann, P., Knackstedt, R., Kuropka, D.: Configurative process modeling - outlining an approach to increased business process model usability. In: Proceedings of the 15th Information Resources Management Association Information Conference (2004)
3. Dreiling, A., Rosemann, M., van der Aalst, W.M.P., Heuser, L., Schulz, K.: Model-based software configuration: patterns and languages. EJIS 15(6), 583–600 (2006)
4. Dumas, M., van der Aalst, W.M.P., ter Hofstede, A.H.M.: Process-Aware Information Systems: Bridging People and Software through Process Technology. Wiley Interscience (2005)
5. Dwyer, M.B., Avrunin, G.S., Corbett, J.C.: Patterns in property specifications for finite-state verification. In: Proceedings of the 21st International Conference on Software Engineering, ICSE 1999, pp. 411–420. ACM (1999)

6. 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: Halpin, T., Krogstie, J., Nurcan, S., Proper, E., Schmidt, R., Soffer, P., Ukor, R. (eds.) BPMDS 2009 and EMMSAD 2009. LNBIP, vol. 29, pp. 353–366. Springer, Heidelberg (2009)
7. Giannakopoulou, D., Havelund, K.: Automata-based verification of temporal properties on running programs. In: ASE, pp. 412–416. IEEE Computer Society (2001)
8. Gottschalk, F.: Configurable Process Models. Ph.D. thesis, Eindhoven University of Technology, The Netherlands (December 2009)
9. Gottschalk, F., van der Aalst, W.M.P., Jansen-Vullers, M., La Rosa, M.: Configurable Workflow Models. International Journal on Cooperative Information Systems 17(2) (2008)
10. Lamport, L.: The temporal logic of actions. ACM Trans. Program. Lang. Syst. 16(3), 872–923 (1994)
11. Pesic, M.: Constraint-Based Workflow Management Systems: Shifting Controls to Users. Ph.D. thesis, Beta Research School for Operations Management and Logistics, Eindhoven (2008)
12. Pichler, P., Weber, B., Zugal, S., Pinggera, J., Mendling, J., Reijers, H.A.: Imperative versus Declarative Process Modeling Languages: An Empirical Investigation. In: Daniel, F., Barkaoui, K., Dustdar, S. (eds.) BPM Workshops 2011, Part I. LNBIP, vol. 99, pp. 383–394. Springer, Heidelberg (2012)
13. Rosemann, M., van der Aalst, W.M.P.: A configurable reference modelling language. Information Systems 32, 1–23 (2007)
14. Sadiq, S.W., Orlowska, M.E., Sadiq, W.: Specification and validation of process constraints for flexible workflows. Information Systems 30(5), 349–378 (2005)
15. Schumm, D., Leymann, F., Streule, A.: Process viewing patterns. In: EDOC, pp. 89–98 (2010)
16. Schunselaar, D.M.M.: Configurable Declare. Master's thesis, Eindhoven University of Technology (2011),
    http://alexandria.tue.nl/extra1/afstversl/wsk-i/schunselaar2011.pdf
17. Schunselaar, D.M.M., Maggi, F.M., Sidorova, N.: Patterns for a Log-Based Strengthening of Declarative Compliance Models. In: Derrick, J., Gnesi, S., Latella, D., Treharne, H. (eds.) IFM 2012. LNCS, vol. 7321, pp. 327–342. Springer, Heidelberg (2012)
18. Schunselaar, D.M.M., Verbeek, E., van der Aalst, W.M.P., Raijers, H.A.: Creating Sound and Reversible Configurable Process Models Using CoSeNets. In: Abramowicz, W., Kriksciuniene, D., Sakalauskas, V. (eds.) BIS 2012. LNBIP, vol. 117, pp. 24–35. Springer, Heidelberg (2012)
19. 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)