Configurable Declare: Designing Customisable Flexible Models

Dennis M. M. Schunselaar, Fabrizio M. Maggi *, and Natalia Sidorova

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}@tue.nl

Abstract. Keywords:

1 Introduction

Process-aware information systems [3], 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 a hart of such systems since they define the flow of task executions. Traditionally, the languages used for process modeling are *imperative* languages, since they are very appropriate for describing well-structured processes with a predefined flow. At the same time, imperative 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 protocols, the process flow cannot be completely predefined, and the imperative way of modeling becomes impossible.

Unlike imperative languages, specifying what should be done, *declarative* languages specify which constraints may not be violated, and therefore they allow for comprehensible descriptions of process 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. In the last years, a number of declarative process modeling languages were developed [] and proven to be more suitable for certain application domains than imperative languages [4]. However, since declarative process modeling languages attracted the attention of the research community at the later stage, when imperative languages were already massively used, there are still serious gaps in the domain of declarative process modeling 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

* This research has been carried out as a part of the Poseidon project at Thales under the responsibilities of the Embedded Systems Institute (ESI). The project is partially supported by the Dutch Ministry of Economic Affairs under the BSIK program.

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 every municipality. A one-size-fits-all approach with full standartisation of processes is however often inappropriate, for these organisations might (believe to) have good reasons for structuring their processes the way they do.

Configurable process models were introduced to solve the aforementioned problems [6,5]. 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. Examples of configurable process modelling languages are *C-SAP WebFlow*, *C-BPEL*, and *C-YAWL* [5].

To the best of our knowledge, the only kind of existing configurable process models are *imperative* models. In this paper, we study in which way configurability in the declarative context is different from configurability in the imperative context. For this purpose we consider the example declarative language *Declare* [9,10] in which constraints are LTL-formulas evaluated on the 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, the configuration options we include in *Configurable Declare* are (1) *hiding* an event, (2) *removing a constraint*, and (3) *substituting a constraint* by some constraint from a set of alternatives defined in the configurable model.

Like many other 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 behavior of a model in which an event is hidden should remain the same as it was modulo this event. In the case of *Declare*, where process behavior is considered to be defined by the set of event traces, hiding 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 the action from the model. 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 option to a configured model.

Since some combinations of configuration options might lead to uninteresting or undesirable models, we introduce meta-constraints, which are defined as logical expressions over available configuration choices, e.g. "if event A is hidden, then event B is not hidden". Similarly to questionnaires used for choosing

configuration options when using imperative process modeling languages, meta-constraints allow for support of a user-friendly way of configuring models using questionnaires.

We have developed a tool **name of the tool?** ... and use this tool for a case study based on business process models from several Dutch municipalities. These models represent the production of an excerpt from the civil registration. We developed a *Configurable Declare* model from which the municipalities' models can be derived.

This paper is structured as follows: In Section 2, we give an overview of related work. *Configurable Declare* is explained in Section 3 along with a further explanation of the configuration patterns listed in Section 2. Section 3 gives a brief introduction to Declare. In Section 4, we introduce *Configurable Declare* and in Section 5, we explain how to derive a *Declare* model from a *Configurable Declare* model and chosen configuration options. Section **??** presents the case study. Finally, in Section 6 we draw some conclusions and discuss directions for future work.

2 Related Work

http://www.springerlink.com/content/t0118027545m5172/fulltext.pdf contains some references.

Configurable process models have been defined for some imperative modelling languages, e.g. *C-SAP WebFlow*, *C-BPEL*, *C-YAWL* [5], and *C-EPC* [11]. However, to our knowledge, no configurable version has been defined for a declarative modelling language.

Imperative configurable process models have to support a number of operations, e.g. the removal of a task from the process model. Some patterns have been identified to support these operations [2,12,1,5]. We want that *Configurable Declare* supports the identified patterns which entail the model itself, i.e. which reason about the control-flow instead of, for instance, the different views on the model.

***** Todo: We can even remove this section and merge it with the introduction *****

3 Background

A process can be described by using different types of modeling languages. Modeling languages can be classified according to two main categories: procedural and declarative. A procedural model is a "closed world", i.e., it explicitly specifies all the acceptable sequences of activities in the process and everything that is not mentioned is forbidden. These models can be used to provide a high level

of support to participants that, during the process execution, must simply follow one of the allowed sequences in the model. Therefore, this type of models is optimal in environments that are stable and where decisions about how to work are made centrally (e.g., a reviewer in a procedure to review a paper). In contrast, a declarative model describes a process as a list of properties that must be satisfied during the process execution. A declarative model is an "open world" where everything is allowed unless it is explicitly forbidden. 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). At the same time, their representation remains compact.

3.1 Declare: A User-Friendly Declarative Language

Figure 1 shows a simple *Declare* model to describe the process for requesting an excerpt from the civil registration in a Dutch municipality. This model is part of a real case study conducted in collaboration with some Dutch municipalities. The different municipalities model this process in different, but very similar ways. Figure 1 is one of these process models that we use to introduce the *Declare* language. We refer the reader to [8] for a complete description of *Declare*.

The *Declare* model in Fig. 1 involves nine *events* (depicted as rectangles, e.g., Send payment request) and nine *constraints* (shown 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 behaviors, implicitly identifying the acceptable sequences of events that comply with (all of) them.

The model shows that the first task to be performed in the process must be Fill in e-form, as indicated by the *init* constraint associated to this event. After that, it is possible to perform Send payment request, but only after having performed Determine administrative expenses (it is necessary to exactly evaluate the total amount of administrative expenses before formulating a payment request). This is indicated by the *precedence* constraint between the two events. A *succession* constraint is the combination of *precedence* and *response*. For instance, when Fill in payment information is executed Process payment must be eventually executed but Process payment cannot complete before Fill in payment information has completed. One of tasks Stop (that stops the procedure) or Fill in payment information must be performed in the process but not both, as indicated by the *exclusive choice* between such two events. Finally, the semantics of 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.

Every *Declare* constraint can be expressed via an automaton, which only accepts valid execution sequences (the language for that automaton). Consider, for instance, the automaton for the $response(A, B)$ constraint in Figure 2. In this automaton, the initial state (indicate with the small arrow) is an accepting state. Whenever we perform an event different from A (i.e. the $\neg A$ clause on

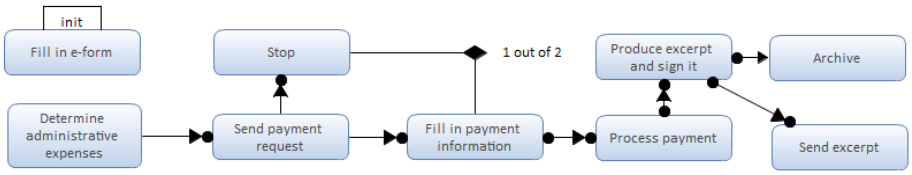


Fig. 1. Example Declare model.

the self-loop), we remain in the initial accepting state. If we perform an A, we (eventually) have to perform a B. Therefore, if we perform an A, in the automaton we move to a non-accepting state and we move back to the initial accepting state only when B is executed.

Formally, a *Declare* model can be defined as follows:

Definition 1. *Let $\mathfrak{E}$ be the universe of events, and $\mathfrak{C}$ the universe of all constraints expressed in LTL-formulas, then a Declare model is a pair $(\mathcal{E}, \mathcal{C})$, s.t. $\mathcal{E} \subseteq \mathfrak{E}$ is a set of events in the model, and $\mathcal{C} \subseteq \mathcal{C}_{\mathcal{E}} \subseteq \mathfrak{C}$ is a set of constraint in the model, where every $c \in \mathcal{C}_{\mathcal{E}}$ only reasons about events in $\mathcal{E}$.*

4 Configurable Declare

In this section, we define *Configurable Declare*, a language to describe configurable *Declare* models. We first describe how we extend *Declare* to obtain *Configurable Declare*. Afterwards, we present the steps for defining a configuration on a given *Configurable Declare* model.

4.1 Extending Declare to Configurable Declare

The core mechanism of the *Configurable Declare* consists of annotating some parts of a *Declare* model thus allowing users to "configure" these parts according to their specific needs. In particular, it is possible to configure a *Configurable Declare* model by (1) hiding an event (Fig. 3(a)), (2) omitting a constraint (Fig. 3(b)), and (3) substituting a constraint by an alternative constraint (Fig. 3(c)).

In *Configurable Declare*, events and constraints are made configurable through annotations. We call these annotations *configuration options*. In particular, a

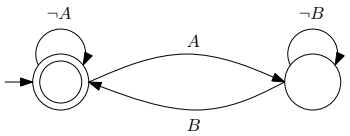


Fig. 2. The automaton for the constraint $response(A, B)$

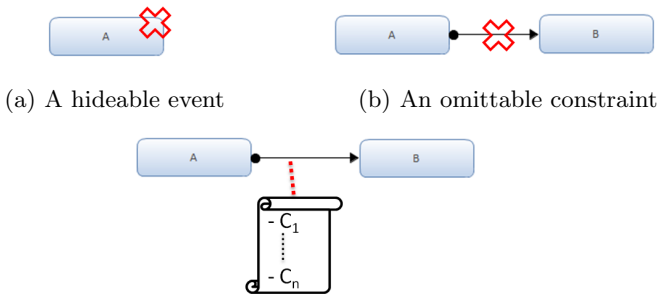
configurable event is annotated with the attribute *hidable* which denotes whether this event can be hidden in the target model. If the event will be hidden, the model will not specify any constraint on it. A *configurable constraint* is annotated with the attribute *omittable* which indicates whether this constraint can be removed from the model. Furthermore, each configurable constraint is also associated to a (possibly empty) list containing all the possible alternative constraints it can be substituted by. Note, that a configurable constraint can be both optional (i.e., *omittable* is true) and substitutable, (i.e. *substitutableBy* yields a non-empty list).

A *Configurable Declare* model is, therefore, defined as follows:

Definition 2. *A **Configurable Declare model*** $(M = (\mathcal{E}, \mathcal{C}, hidable, omittable, substitutableBy))$ *is an extension of a Declare model obtained by annotating events and constraints with **configuration options** specified through the functions: hidable* $: \mathcal{E} \rightarrow \mathbb{B}$ *denoting whether an event can be hidden, omittable* $: \mathcal{C} \rightarrow \mathbb{B}$ *denoting whether a constraint can be omitted, and substitutableBy* $: \mathcal{C} \rightarrow 2^{\mathcal{C}_\mathcal{E}}$ *denoting the list of alternatives for a constraint (remember that every* $c \in \mathcal{C}_\mathcal{E}$ *only reasons about events in* $\mathcal{E}$*).*

4.2 Configuring Configurable Declare

When a configurable model has been designed, this model can be configured in different ways by applying a configuration on it. A configuration of a *Configurable Declare* model is defined as a set of mappings between configuration options (specified through the functions *hidable*, *omittable* and *substitutableBy*) and configuration choices that can be specified through three different functions: *hiding* $: \mathcal{E}' \rightarrow \mathbb{B}$, where $\mathcal{E}' = \{a \in \mathcal{E} | hidable(a) = true\}$, that associates to each configuration option $hidable(a) = true$ the configuration choice *true* or *false* indicating whether a hidable event must be hidden in the target *Declare* model; *omitting* $: \mathcal{C}' \rightarrow \mathbb{B}$, where $\mathcal{C}' = \{c \in \mathcal{C} | omittable(c) = true\}$,



(a) A hideable event

(b) An omittable constraint

(c) A constraint substitutable by $c_1, \cdots c_n$

Fig. 3. The graphical representation of the different configuration options

that associates to each configuration option $omittable(c) = true$ the configuration choice *true* or *false* indicating whether an omittable event must be omitted in the target *Declare* model; and *substitution* on $\mathcal{C}' \rightarrow \mathcal{C}_{\mathcal{E}}$, where $\mathcal{C}' = \{c \in \mathcal{C} | substitutableBy(c) \neq \emptyset\}$ and every $c \in \mathcal{C}_{\mathcal{E}}$ only reasons about events in $\mathcal{E}$, that associates to each configuration option $substitutableBy(c) \neq \emptyset$ the configuration choice $c' \in substitutableBy(c) \cup \{c\}$ indicating that c must be replaced by c' in the target *Declare* model. However, these functions still allow for the substitution of an omitted constraint, therefore, we limit the domain of *substitution* to $\mathcal{C}' = \{c \in \mathcal{C} | substitutableBy(c) \neq \emptyset \wedge omittable(c) \Rightarrow \neg omitting(c)\} \cup \{c\}$.

***** Fixed: what if I want to keep c which is substitutable in the target model? *****

Therefore, a configuration can be defined as follows:

Definition 3. *A* configuration $C = (hiding, omitting, substitution)$ *of a Configurable Declare model* $(M = (\mathcal{E}, \mathcal{C}, hidable, omittable, substitutableBy))$*, is a partial mapping from configuration options to* configuration choices *specified through the functions* $hiding : \mathcal{E} \nrightarrow \mathbb{B}$ *denoting whether a hidable event must be hidden in the target model,* $omitting : \mathcal{C} \nrightarrow \mathbb{B}$ *denoting whether an omittable constraint must be omitted in the target model, and* $substitution : \mathcal{C} \nrightarrow \mathcal{C}_{\mathcal{E}}$ *denoting, for each substitutableBy constraint, the alternative constraint (it must be substituted by) in the target model.*

In order to guide the creation of a configuration, meta-constraints are defined for a *Configurable Declare* model. Let $\mathfrak{F}$ be the set of all possible logical formulas, then the meta-constraints are a set $\mathcal{F}$, s.t. $\mathcal{F} \subseteq \mathcal{F}_{\mathcal{EC}} \subseteq \mathfrak{F}$ where $\mathcal{F}_{\mathcal{EC}}$ is the set of logical formulas over configuration choices for $\mathcal{E}$ and $\mathcal{C}$.

For instance, the meta-constraint to specify that "if constraint c is not omitted, then event B must be hidden" is encoded as $\neg omitting(c) \Rightarrow hiding(B)$. The meta-constraint to express that "when event A is hidden, option c_k must be chosen for constraint c" is encoded as $hiding(A) \Rightarrow substitution(c) = c_k$. Using these meta-constraints, we can prevent the creation of a configuration which, if applied to the *Configurable Declare* model, yields a *Declare* model which does not adhere to some predetermined criteria (e.g., laws or regulations) or is inconsistent.

Figure 4 depicts the *Configurable Declare* model we have built in the context of our case study. This model entail the request for an excerpt from the civil registration within a municipality. The parts of the model that are not annotated with a configuration option have to be performed by all municipalities. This entails the request itself and partially taking care of the payment. Then, the municipalities can choose how to process the payment, and whether they want to archive the request. Finally, all municipalities have to produce the excerpt and send this excerpt to the customer. The municipalities have the option to hide some events as well as remove and substitute constraints.

Table 1. The configurations for the municipalities (mun.)

Configuration Point	mun. A	mun. B	mun. C
Send to DMS department	Hide	Hide	Keep
Indicate already paid	Hide	Hide	Keep
Process payment	Keep	Keep	Hide
Print request	Hide	Keep	Hide
Archive	Keep	Hide	Keep
response(Send excerpt, Archive)	Omit	Keep	Omit
precedence(Produce excerpt and sign it, Archive)	Subst. by *response*	Keep	Subst. by *response*

Table 1 contains the different configurations for our case study.

In the next section we describe how to obtain a *Declare* model by applying a configuration to a *Configurable Declare* model.

Notice that defining a valid configuration (i.e., consistent with all the meta-constraints) on a *Configurable Declare* model is an error-prone and time-consuming task. Furthermore, domain-experts do not have in general enough expertise to define a proper configuration. To address this issue, we can apply a general approach based on questionnaires as introduced in [7].

In this approach, each question of a questionnaire is linked to one or more decision points in a configurable model. Decision points are elements (events and constraints in our case) which can be configured. The answers specified by the user in a questionnaire determine which events must be hidden from the model and which constraints must hold between them. In this way, by answering the questionnaire, users automatically configure the *Configurable Declare* model according to their specific needs.

5 Obtaining the Declare model

For each type of configuration choice (hiding an event, omitting a constraint and substituting a constraint by an alternative constraint) in a configuration,

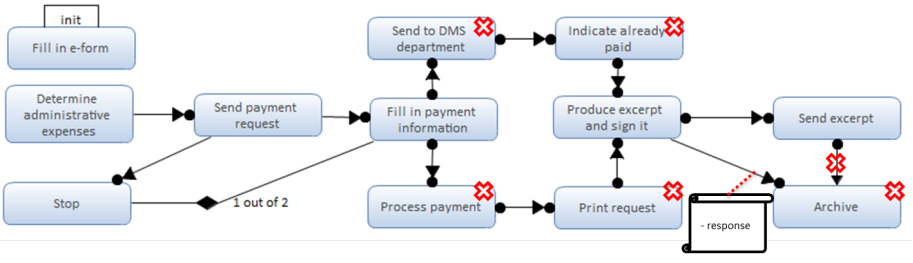


Fig. 4. The *Configurable Declare* model used for the case study

we define transformation rules. These transformation rules allow users to apply a configuration to a *Configurable Declare* model, thus obtaining a *Declare* model. The transformation rules for removing and substituting a constraint are trivial, i.e., we simply remove or substitute that constraint without performing any additional actions. In contrast, a transformation rule for hiding an event is less trivial. Therefore, in the following, we only elaborate on this.

5.1 Hiding an event

The main issue related to hiding an event is that, when hiding an event, we have to maintain implicit constraints. Implicit constraints are constraints that hold in the model but are implicitly encoded by using other (explicit) constraints. Consider, for instance, the model depicted in Fig. 5 where we have two *response* constraints: $response(A, B)$ (every A is eventually followed by B) and $response(B, C)$ (every B is eventually followed by C). These constraints implicitly encode the constraint $response(A, C)$ that states that every A is eventually followed by C.

The transformation rules we define, maintain implicit constraints like this and, if necessary, make them explicit in the model. For instance, in Fig. 5, we add the constraint $response(A, C)$, if we hide event B in the model. Due to space restrictions, we only present here a subset of the entire set of possible implicit constraints (Table 2). We refer the interested reader to [13] for a complete overview about them. In order to show the correctness of the rules, we deduce the implicit constraints by applying τ-abstraction on finite state automata.

Formally, the issue deriving from the existence of implicit constraints in a *Configurable Declare* model can be synthesised as follows. Let $\mathcal{M}$ be the original model and $\mathcal{M}$' be the model where some event A has been hidden, then we want that $\mathcal{L}_{\mathcal{M}}^{A \leftarrow \tau} \supseteq \mathcal{L}_{\mathcal{M}'}$, i.e., the language of model $\mathcal{M}$, after applying τ abstraction on A, should contain the language of model $\mathcal{M}$' (where A has been hidden).

***** Todo: do we want to prove the equivalence? *****

For instance, if $M = response(A, B) \wedge response(B, C)$ and $M' = response(A, C)$, we show that $\mathcal{L}_{response(A,B) \wedge response(B,C)}^{B \leftarrow \tau} \supseteq \mathcal{L}_{response(A,C)}$

Fig. 6, shows the automaton for the language: $\mathcal{L}\left(response(A, B) \wedge response(B, C)\right)$. Applying (partial) τ-abstraction yields the automaton in Fig. 7. By removing

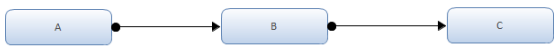


Fig. 5. An example model in which we have implicitly a *response* between A and C

Table 2. A subset of the language inclusions after the τ-abstraction of B

$$
\begin{array}{ll}
\mathcal{L}^{B\leftarrow\tau}(response(A,B) \wedge response(B,C)) & \supseteq \mathcal{L}(response(A,C)) \\
\mathcal{L}^{B\leftarrow\tau}(response(A,B) \wedge succession(B,C)) & \supseteq \mathcal{L}(response(A,C)) \\
\mathcal{L}^{B\leftarrow\tau}(precedence(A,B) \wedge precedence(B,C)) & \supseteq \mathcal{L}(precedence(A,C)) \\
\mathcal{L}^{B\leftarrow\tau}(precedence(A,B) \wedge succession(B,C)) & \supseteq \mathcal{L}(precedence(A,C)) \\
\mathcal{L}^{B\leftarrow\tau}(succession(A,B) \wedge response(B,C)) & \supseteq \mathcal{L}(response(A,C)) \\
\mathcal{L}^{B\leftarrow\tau}(succession(A,B) \wedge precedence(B,C)) & \supseteq \mathcal{L}(precedence(A,C)) \\
\mathcal{L}^{B\leftarrow\tau}(succession(A,B) \wedge succession(B,C)) & \supseteq \mathcal{L}(succession(A,C))
\end{array}
$$

the τ-transition, we limit the behaviour, thus we obtain a subset of the language as defined by the automaton without the substitution of B by τ.

In this way, we obtain the first inclusion in Table 2. Similarly, we can apply the same approach to demonstrate the other inclusions.

Whenever a configuration choice leads to a transformation rule where some implicit constraints are involved the inclusions listed in Table 2 must be taken into account. In some cases, e.g. $response(A,B) \wedge response(C,B)$, we do not have any implicit constraints after the hiding of B and we can hide B straightforward without introducing any implicit constraints.

***** Todo: can we limit for now the demonstration to one direction (inclusion)? *****

5.2 Methodology

Using the aforementioned transformation rules for each configuration choice defined in the *Configurable Declare* model, we can convert a *Configurable Declare*

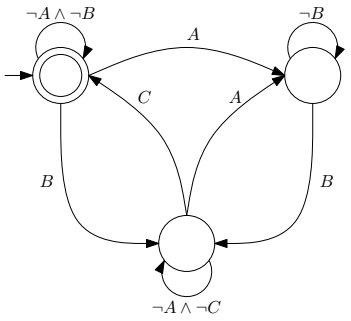


Fig. 6. The automaton for $\mathcal{L}\left(response(A,B) \wedge response(B,C)\right)$

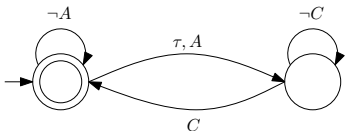


Fig. 7. The intermediate automaton after (partially) applying τ-abstraction

model with a configuration into a *Declare* model (Algorithm 1). The first thing to do in this procedure is to apply the transformation rules on constraints. In particular, we remove the constraints that must be omitted and transform the constraints that must be substituted by a different one (as indicated in the configuration). After having updated the set of constraints, we operate on events. In particular, we remove the events as indicated in the configuration and, in this case, we apply the transformation rules to maintain the implicit constraints in the configured model. If an implicit constraint is not encoded in to model, due to the removal of an event e, we add this constraint to the model. Finally, we remove the omitted events and all constraints associated with these events from the model.

Algorithm 1: The pseudo code for transforming a configured configurable *Declare* model into a *Declare* model

TransformToDeclare($\mathcal{M}_{conf}$, $\mathcal{M}_{decl}$)
Input: $\mathcal{M}_{conf}$ the configured configurable *Declare* model
Output: $\mathcal{M}_{decl}$ the *Declare* model
(1) $\mathcal{M}_{temp} \leftarrow \mathcal{M}_{conf}$
(2) **foreach** constraint $c \in \mathcal{M}_{temp}$
(3) **if** c has to be removed **then**
(4) remove c from $\mathcal{M}_{temp}$
(5) **else if** c is substituted by a different constraint c' **then**
(6) substitute c by c'
(7) **foreach** event $e \in \mathcal{M}_{temp}$
(8) **if** e is configured to be removed
(9) $\mathcal{C} \leftarrow$ all implicit constraints which have to be made explicit after the removal of e using the defined rules
(10) add all constraints from $\mathcal{C}$ to $\mathcal{M}_{temp}$
(11) Remove all events from $\mathcal{M}_{temp}$ which are omitted and all constraints associated with these events
(12) **return** $\mathcal{M}_{temp}$

In our case study, after having defined our configurations (Table 1), we apply the methodology mentioned above. We show how it works with the configuration for municipality A. At the same way we can proceed for the other two configurations.

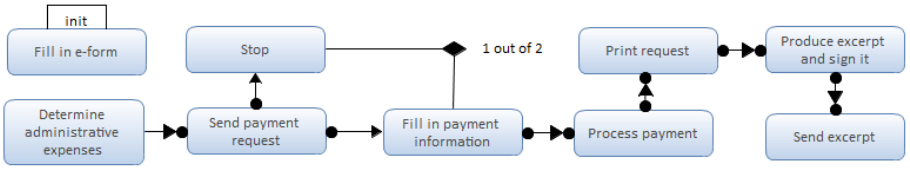


Fig. 8. Output model B

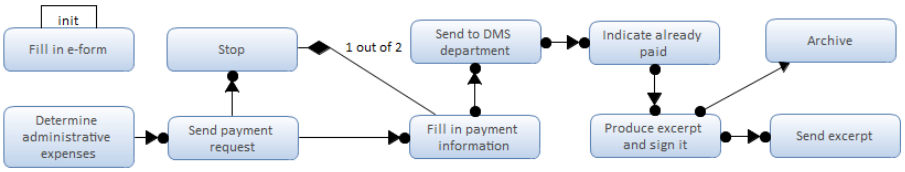


Fig. 9. Output model C

First, we remove the *response* between Send excerpt and Archive, afterwards, we substitute the *precedence* between Produce excerpt and sign it and Archive by a *response* constraint. Finally, we can compute the implicit constraints which might be made explicit due to the removal of an event.

When removing Print request, we obtain a *succession* constraint between Process payment and Produce excerpt and sign it. Since this *succession* is not present in the model, we add this constraint to the model.

When removing Send to DMS department and Indicate already paid, the order in which we remove these events does not matter. Therefore, we choose to first remove Send to DMS department and, afterwards, remove Indicate already paid. When removing Send to DMS department, we introduce a *succession* between Fill in payment information and Indicate already paid. After removing Indicate already paid, we introduce a *succession* between Fill in payment information and Produce excerpt and sign it. This constraint is already implicitly encoded in the model, by the *succession* between Fill in payment information and Process payment and between Process payment and Produce excerpt and sign it. Hence we do not add the *succession* between Fill in payment information and Produce excerpt and sign it to the model.

By removing the omitted events and constraints, and by substituting constraints by different constraints, we obtain the *Declare* model as depicted in Fig. 1.

The *Declare* models we obtain, corresponding to municipalities B and C, are depicted in Fig. 8 and 9.

6 Conclusion

We have shown an approach for transforming a *Configurable Declare* model with a configuration into a *Declare* model. Unfortunately, the *Configurable Declare* model has to be defined by hand, which is error-prone and time-consuming. Currently we are working on an approach to automate the creation of the *Configurable Declare* model, so given a set of *Declare* models, obtain a *Configurable Declare* model. On this *Configurable Declare* model, we have to be able to define a set of configurations which, after applying them, yield the original set of *Declare* models. The merger of different *Declare* models into a single *Configurable Declare* model is achieved by taking the union of the different constraints. Using the implicit constraints defined earlier, we can minimise the total amount of constraints in the *Configurable Declare* model.

The deduction of a configuration entailed either defining the configuration by hand or by using a questionnaire. The next step is deducing a configuration for a *Configurable Declare* model, if it exists, using a log. From the log, we mine a *Declare* model using the *Declare* miner []. Using the approach by [14], we can remove uninteresting constraints, i.e. constraints which cannot be invalidated because either an event never occurs or a stronger constraint is valid.

From this mined model, we determine the events which should be removed, i.e. *hiding* in our configuration. If one of the aforementioned events cannot be hidden, i.e. *hidable* is false, we cannot deduce the configuration. Determining whether the constraints can be expressed is less trivial, since we might need to prevent the deduction of implicit constraints by removing some constraints. By storing which constraints were used for deducing implicit constraints, we can determine which constraints have to be omitted to prevent the deduction of this implicit constraint.

***** Todo: is this enough or do we need to go more in depth?, I might have a nice but messy picture *****

***** Todo: Automatic validation of a questionnaire *****

We can automatically validate a questionnaire by encoding the different options in logical formulas. We want to validate the following formula: *encoding of the questionnaire* $\Rightarrow$ *reference model*. Where the *reference model* is the minimal set of constraints which have to hold in the eventual *Declare* model. For instance, let our questionnaire consist of yes/no-questions $q_1, \cdots q_n$ and they are all reason about a single configuration option and let $c_1, \cdots c_n$ be our set of configuration options, then we have to validate the following: $((q_1 \wedge c_1) \vee \neg q_1) \wedge \cdots \wedge ((q_n \wedge c_n) \vee \neg q_n) \wedge$ *dependencies between questions* $\Rightarrow$ *reference model*. These dependencies are constraints on the different questions, for instance, $q_1 \Rightarrow \neg q_5$.

Acknowledgements. This research has been carried out within the *CoSeLoG* project (http://www.win.tue.nl/coselog/).

References

1. 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 Associatino Information Conference. New Orleans (2004)
2. Dreiling, A., Rosemann, M., van der Aalst, W., Heuser, L., Schulz, K.: Model-based software configuration: Patterns and languages. European Journal of Information Systems 15, 583–600 (2006)
3. Dumas, M., Aalst, W., Hofstede, A.: Process-Aware Information Systems: Bridging People and Software through Process Technology (2005)
4. Fahland, D., Lbke, 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., Aalst, W., Mylopoulos, J., Rosemann, M., Shaw, M.J., Szyperski, C., Aalst, W., Mylopoulos, J., Rosemann, M., Shaw, M.J., Szyperski, C. (eds.) Enterprise, Business-Process and Information Systems Modeling, Lecture Notes in Business Information Processing, vol. 29, pp. 353–366. Springer Berlin Heidelberg (2009), http://dx.doi.org/10.1007/978-3-642-01862-6_29, 10.1007/978-3-642-01862-6_29
5. Gottschalk, F.: Configurable Process Models. Ph.D. thesis, Eindhoven University of Technology, The Netherlands (December 2009)
6. Gottschalk, F., van der Aalst, W.M.P., Jansen-Vullers, M., Rosa, M.L.: Configurable workflow models. International Journal on Cooperative Information Systems 17(2) (2008)
7. La Rosa, M., Lux, J., Seidel, S., Dumas, M., ter Hofstede, A.H.M.: Questionnaire-driven Configuration of Reference Process Models. Advanced Information Systems Engineering 4495, 424–438 (2007), http://dx.doi.org/10.1007/978-3-540-72988-4_30
8. Pesic, M.: Constraint-Based Workflow Management Systems: Shifting Controls to Users. Ph.D. thesis, Beta Research School for Operations Management and Logistics, Eindhoven (2008)
9. Pesic, M., van der Aalst, W.: A declarative approach for flexible business processes management. In: Eder, J., Dustdar, S. (eds.) Business Process Management Workshops, Lecture Notes in Computer Science, vol. 4103, pp. 169–180. Springer Berlin / Heidelberg (2006), http://dx.doi.org/10.1007/11837862_18, 10.1007/11837862_18
10. Pesic, M., Schonenberg, H., van der Aalst, W.: Declare: Full support for loosely-structured processes. In: Enterprise Distributed Object Computing Conference, 2007. EDOC 2007. 11th IEEE International. p. 287 (2007)
11. Rosemann, M., van der Aalst, W.M.P.: A configurable reference modelling language. Inf. Syst. 32, 1–23 (March 2007), http://dl.acm.org/citation.cfm?id=1221586.1221839
12. Schumm, D., Leymann, F., Streule, A.: Process viewing patterns. In: Proceedings of the 14th IEEE International EDOC Conference, EDOC 2010, 25–29 October 2010, Vitoria, Brazil. pp. 89–98. IEEE Computer Society (2010)

13. Schunselaar, D.M.M.: Configurable Declare. Master's thesis, Eindhoven University of Technology (2011), http://alexandria.tue.nl/extra1/afstversl/wsk-i/schunselaar2011.pdf
14. Schunselaar, D.M.M., Maggi, F.M., Sidorova, N.: Do My Constraints Constrain Enough? - Patterns for Strengthening Constraints in Declarative Compliance Models. Tech. rep., BPMcenter.org (2011), bPM Center Report BPM-11-15