Towards Customer-Individual Configurations of Business Process Models

Michael Becker and Stephan Klingner

Department of Business Information Systems, University of Leipzig, Germany
{mbecker,klingner}@informatik.uni-leipzig.de

Abstract. Nowadays business process models are a common approach to describe and analyse existing business processes and to create new processes in a structured way. However, with growing complexity of process models there is a lack of comprehensibility. Using existing notations, it is challenging or even impossible to define temporal and logical constraints between process steps that are not directly connected. We demonstrate a declarative approach for representing business processes that allows for configuration, i.e. selection of process steps, based on a component representation. In addition, we present ways to transform a configuration into a procedural process model using BPMN.

Keywords: Business Process Configuration, Service Modelling, Modularisation.

1 Introduction

The growing economical importance of the service sector is associated with an increasing complexity of services. One example of this trend are offers comprising products and services in so called product-service-systems. These developments are accompanied by a growing demand of customer-individual offers. To achieve fundamental economical aims – despite those challenges – an efficient provision of those services is a necessary precondition. To support a productive and standardised service provision, the modelling of services in terms of service engineering is a widely implemented approach. Various IT-based modelling languages allow for a precise description of the process-related aspects of services. But to widen the focus in terms of individualisation, the consideration of configuration-related requirements is also necessary. Hence, this paper proposes a modelling method aiming to fulfill the specific needs of the configuration of services, as presented in various papers before. This encloses the segregation of semantically related process parts in so called modules [1] as well as the description of dependencies between those modules [2]. Therefore, this paper gives a brief introduction of the concept of modelling service modules and their configuration. According to [1], we define that a service module offers a well-defined functionality via precisely described interfaces. Furthermore, a service module can be used for composition and can, therefore, itself be part of a more coarse-grained service module.

I. Bider et al. (Eds.): BPMDS 2012 and EMMSAD 2012, LNBIP 113, pp. 121–135, 2012.

To support a seamless integration of the proposed modelling method in the overall process of service engineering, two additional steps besides segregation and description of dependencies have to be considered. It is possible to extract service components from existing business process models (*extraction*). These components can then be used as a basis for configuration. Furthermore, business process models can be generated based on customer-individual configurations of services (*generation*).

The main benefit resulting from extraction is the reuse of existing business process models as basis for the creation of configurable models. In doing so, the step towards a configurable service portfolio can be simplified. On the other hand, the generation of business process models based on configurations extends the advantages of process models on the level of individualisation. These business process models are defining the specific process according to the customer-individual offer and therefore can be used as the basis for a workflow description. This paper focuses generation of process models.

In summary, the whole course of action to create configurable service models as proposed in this paper consists of four steps. First, it is necessary to specify the unique service components and establish hierarchical dependencies between these components. This can be done either manual or by extracting these modules from existing business process models. This results in the existence of a component model. Second, it is necessary to declare logical (i.e. non-hierarchical) and temporal dependencies between components based on this component model. Temporal dependencies are evaluated to specify the order of activities and their parallelisation potential. These steps are described in the following section 2. As a third step, the configuration of components conforming to their structure and dependencies is conducted. This configuration is usually established in collaboration with customers, resulting in a customer-individual offer. The configuration is similar to variants in Product Line Engineering according to [3], i.e. a complete configuration can be understood as a specific variant.

Fourth and finally, this customer specific configuration is transformed into a workflow representation. This workflow model can be imported into a workflow management system to guide the process. Configuration and transformation are described in section 3. Following the explanation of the component definition and configuration we depict related work in section 4. This paper concludes with future research directions and an evaluation of the approach in section 5.

2 Defining Component Models

In this section we formalise the component model representing hierarchical process elements. Based on this formalisation, it is possible to derive actual configurations of a process and verify this configuration. The component model is specified using first-order logic. Though this formalisation requires additional initial effort to specify component models, it provides two fundamental benefits. First, it allows to define the used concepts in unambiguous way due to the formal

defined semantics of first-order logic. Second, and even more important for practical applications, it allows for easy extension and adaption to domain specific facts. Due to space limitation, we present only the most relevant concepts of our component model. Further details and formalisations are specified in [4].

A component model is represented as a 6-tuple $M = (C, K, G, card, L, T)$:

- C is a finite, non-empty set of *components*,
- K is a finite, non-empty set of *connectors*,
- $G \subseteq (C \times K) \cup (K \times C) \cup (K \times K)$ is a set of arcs constituting an acyclic *configuration graph* representing *hierarchical dependencies*,
- $card : K \to \mathcal{P}(\mathbb{N} \times \mathbb{N})$ is a mapping from connectors to *cardinalities*,
- L is a finite set of *logical dependencies*,
- T is a finite set of *temporal dependencies*.

In the following subsections, we provide further details for the individual constituents of the component model based on a simplified example inspired by a real world application. The example describes services around assembly and maintenance of photovoltaics installations. An organisation provides installation services where customers can choose between delivery with self-assembly and delivery with assembly services. However, if users chose assembly services there are two constraints to satisfy. First, customers must obtain delivery service, too. Second, assembly can only be executed after delivery. For existing photovoltaics installations, customers can select maintenance services consisting of on-site maintenance, remote maintenance, and cleaning. Finally, it is possible to obtain monitoring services for evaluating the performance of the photovoltaics installation. Monitoring consists of recording, customer-specific performance analysis, and comparison with other installations. Due to hardware requirements, customers choosing performance analysis have to chose remote maintenance, too. Finally, comparison services needs recording services.

Based on the description of the example, it is possible to identify process components. To shorten formulae presented in the following, we use component identifiers. The process consists of the following components: *Photovoltaics* (C_1) representing the complete process, *Installation* (C_2), *Maintenance* (C_3), and *Monitoring* (C_4) to represent the three main services. *Delivery* (C_5) and *Assembling* (C_6) are installation services. *On-Site Maintenance* (C_7), *Cleaning* (C_8), and *Remote Maintenance* (C_9) are maintenance services. *Comparison* (C_{10}), *Analysis* (C_{11}), and *Recording* (C_{12}) are monitoring services. Thus, we have the set of components $C = \{C_1, C_2, C_3, C_4, C_5, C_6, C_7, C_8, C_9, C_{10}, C_{11}, C_{12}\}$. Fig. 1 on page 127 gives a visualisation of the component model containing all details described in the following sections.

2.1 Hierarchical Dependencies

The main objective of the component model is to describe complex processes using single, less complex subcomponents. This is achieved by using the configuration graph that contains hierarchical dependencies between components.

Decomposing components into more fine grained subcomponents represents process refinement. We use connectors to specify the type of hierarchic relation between components. For configuration reasons, we prohibit direct linking of components, i.e. components can only be linked with each other using connectors. Therefore, we have the set of connectors $K = \{K_1, K_2, K_3, K_4\}$ with K_1 linking the overall component C_1 with the main services C_2, C_3, and C_4. K_2, K_3, and K_4 link the respective subcomponents with their children. This results in the following configuration graph G.

$$G = \{(C_1, K_1), (K_1, C_2), (K_1, C_3), (K_1, C_4), (C_2, K_2), (C_3, K_3), (C_4, K_4),$$
$$(K_2, C_5), (K_2, C_6), (K_3, C_7), (K_3, C_8), (K_3, C_9), (K_4, C_{10}), (K_4, C_{11}), (K_4, C_{12})\}.$$

A connector can be assigned with an arbitrary amount of cardinalities specifying valid configuration choices using the mapping *card*. Each cardinality specifies the minimal and maximum number of subnodes that needs to be selected during configuration. If a connector is assigned with more than one cardinality, the cardinalities are linked with each other using logical ORs. Therefore, only one of the cardinalities must be satisfied during configuration. In our example, we have to choose at least one subcomponent of every component (e.g. if monitoring is chosen, at least one of comparison, analysis, and recording must be chosen as well). Therefore, we have the cardinalities $card(K_1) = \{(1, 3)\}$, $card(K_2) = \{(1, 2)\}$, $card(K_3) = \{(1, 3)\}$, and $card(K_4) = \{(1, 3)\}$.

2.2 Logical Dependencies

The specified graph with different types of connecting nodes and cardinalities defines the dependencies between components that are directly interrelated by the given graph. Additionally, dependencies have to be specified for components that are neither children nor parents of other components. These dependencies are necessary to make statements like *when choosing component A during a configuration, component X has to be chosen as well* or *when choosing component B during a configuration, component Y must not be chosen*. As these dependencies represent logical restrictions for configuration, they are called *logical dependencies* . Such logical dependencies are necessary to support error-free service configurations [5].

For specifying these dependencies, expressive methods like first-order logic are best suited. As the application of first-order logic is quite complex, typical rules can be specified that are based on first-order logic, but are at the same time applicable for users that are unfamiliar with first-order logic. The following rules in Table 1 are examples that can be used when specifying logical dependencies between service modules.

The dependencies given in Table 1 are *hard* dependencies, i.e. they must be satisfied in valid configurations. Besides this, it is also possible to specify *soft* dependencies, e.g. component A is an *alternative* of component B. For establishing new logical dependencies, it is possible to combine existing ones, e.g. components A and B are *exclusive alternatives* of each other can be defined as

Table 1. Logical dependencies between components

Rule	Formalisation	Explanation
Requirement	$requires : C \rightarrow C$ $requires(A) = B$	Component A requires component B: if component A is chosen during configuration, component B must be chosen as well.
Dependency	$depends : C \rightarrow C$ $depends(A) = B$	Component A depends on component B: if component B is not chosen during configuration, component A must not be chosen as well.
Prohibition	$prohibits : C \rightarrow C$ $prohibits(A) = B$	Component A prohibits component B: if component A is chosen during configuration, component B must not be chosen and vice versa.

$prohibits(A) = B$ and $alternative(A) = B$. A selection of additional dependencies in the domain of Product-Service-Systems is presented in [2]. To validate the applicability, we have implemented the dependencies as Prolog rules[1]. Based on this representation, it is possible to verify if a configuration (i.e. a set of selected components) satisfies the given hard dependencies of a component model and whether additional soft dependencies are available. The semantics of the given logical dependencies are defined during configuration (see section 3.2).

Three hard logical dependencies exist in the photovoltaics example. If customers choose assembly service (C_6), they have to choose delivery (C_5), too. Comparison (C_{10}) needs recording (C_{12}). Finally, performance analysis (C_{11}) requires remote maintenance (C_9). This results in logical dependencies $L = \{requires(C_6) = C_5, requires(C_{10}) = C_{12}, requires(C_{11}) = C_9\}$. Especially the requirements relation between performance analysis and remote maintenance is notable. In traditional process models it is often only possible to declare relations in one branch, e.g. relations between the subcomponents of maintenance.

Though we provide the opportunity to define logical dependencies this feature should be used sparsely. It both impacts the readability and comprehensibility of component models and adds to the complexity of configuration. As Thum et al. have shown for feature models these dependencies are especially hard to understand in editing models [6]. Some ideas how to eliminate non-hierarchic constraints in the domain of feature models are given in [7,8]. These concepts should be applicable in the domain of process components, too.

2.3 Temporal Dependencies

Since the components in our model represent process activities, it is necessary to specify the possible execution order of these activities, e.g. sequential execution, parallelisation, and synchronisation. Concerning the specified graph, it is, therefore, not enough to display only logical dependencies between components, but also to display temporal dependencies. These temporal dependencies define whether a component has to be performed before or after another component.

[1] `https://sourceforge.net/projects/kpstools/`

Using this information, it will be easier to implement finally the whole process out of the chosen component. The instantiation of a process (specifying which service module has to be executed at which time) has to take into account the specified temporal dependencies.

For keeping the graph as flexible as possible, temporal dependencies can be specified by using a declarative approach (as opposed to a procedural approach). Such approach has been proposed by Aalst and Pesic [9]. The application of the linear temporal logic (LTL) [10] would offer the most flexible and expressive way of specifying the temporal dependencies. Nevertheless, this approach is not applicable for users that are unfamiliar with LTL. Therefore, a set of rules can be specified that covers most of the temporal dependencies. These rules are, on the one hand, understandable for non-professionals and, on the other hand, based on the LTL which allows functionalities like model checking or simulation. Table 2 shows a selection of possible temporal dependencies. It is necessary to note that the exact semantics of temporal dependencies is formalised during configuration (see section 3.3). Furthermore, the examples shown in Table 2 are not independent of each other, i.e. precedence can be specified in terms of succession and vice versa.

Table 2. Temporal dependencies between components

Rule	Formalisation	Explanation
Precedence	$before(A) = B$	In all configurations containing components A and B, it is necessary to execute component B before component A.
Direct Precedence	$iBefore(A) = B$	In all configurations containing components A and B, it is necessary to execute component B directly before component A.
Succession	$after(A) = B$	In all configurations containing components A and B, it is necessary to execute component B after component A.
Direct Succession	$iAfter(A) = B$	In all configurations containing components A and B, it is necessary to execute component B immediately after component A.

In the photovoltaics example one temporal dependency occurs, i.e. before assembling (C_6) the installation has to be delivered (C_5). Therefore, the set T of temporal dependencies is defined as follows: $T = \{before(C_6) = C_5\}$. It is necessary to mention that all components that are not linked with temporal dependencies are independent from each other. That means, they can be executed in parallel, e.g. while performance analysis it is possible to clean the photovoltaics installation. In a complex real world example there would be much more temporal dependencies. For example, cleaning and on-site maintenance may be performed by the same individuals. Therefore, only one of the activities can be executed at one time. Furthermore, installation certainly needs to be completed before maintenance.

2.4 Graphical Representation

For comprehensibility reasons, we provide a set of notational elements for the graphical representation of a component model. Components are depicted as rectangles and connectors as circles. Hierarchical dependencies between these elements are represented using directed arrows. To represent logical dependencies, we use directed, dotted arrows where $A \rightarrow B$ means that selecting component A also requires selecting component B. Temporal dependencies are represented using dashed lines, where $A \rightarrow B$ means that component A must be performed before component B. Fig. 1 shows the photovoltaics example using the defined notational elements.

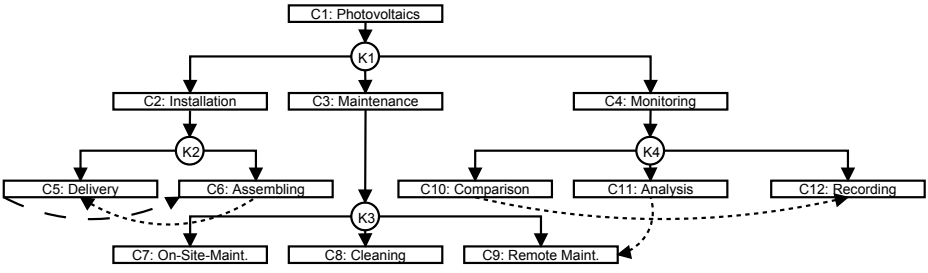


Fig. 1. Component model for a photovoltaics installation

3 Configuration

Using the hierarchical, logical, and temporal dependencies between components, it is possible to generate a customer specific configuration. A configuration is a set of components selected from the given portfolio defined by the component model. Due to the formalisation of the model, it is possible to validate whether selected components fulfill the established dependencies. In this section we use the photovoltaics example introduced in the last section and show how to create configurations and how to establish process models based on a given configuration.

Each configuration consists of three distinct steps. First, components are selected based on customer-specific requirements. The selection is restricted by hierarchical dependencies between components, cardinalities of connectors, and hard logical dependencies between components. Second, soft logical dependencies are evaluated and presented as configuration alternatives. Third, an actual configuration is transformed into a procedural process representation using the temporal dependencies. This representation can for example be imported in workflow management systems to guide through the process.

3.1 Component Selection and Cardinality Evaluation

During configuration, the mapping $s : C \rightarrow \{0, 1\}$ represents whether a component is selected or not. A configuration is defined as the set of selected

components, i.e. the set $Configuration = \{c | c \in C \wedge s(c) = 1\}$ contains all selected components. At the beginning of the configuration process, none of the nodes is selected, i.e. $\forall c \in C : s(c) = 0$.

To define selection and connector semantics during configuration we need to introduce the mapping $p : C \cup K \rightarrow \mathcal{P}(C \cup K)$ to identify postnodes (i.e. succeeding nodes) of a node in the configuration graph. This mapping is defined as $p(n_1) = \{n_2 \in C \cup K : \exists e \in G : e = (n_1, n_2)\}$.

Now we define that succeeding nodes of an unselected node are not selected, too. Thus, we prohibit to select subcomponents without selecting the respective superior component: $\forall n_1 \in C \cup K, \forall n_2 \in p(n_1) : s(n_1) = 0 \rightarrow s(n_2) = 0$.

On the opposite, all succeeding nodes of a component are selected. Since components can only be followed by connectors, we include these connectors in the configuration: $\forall n_1 \in C, \forall n_2 \in p(n_1) : s(n_1) = 1 \rightarrow s_(n_2) = 1$.

Finally, we have to define connector semantics during configuration. A connector is satisfied if there is a number of succeeding nodes selected fulfilling the interval defined by one of the cardinalities. Therefore, we define the set sp that contains all selected succeeding nodes of a connector k.

$$\forall k \in K : sp(k) \subseteq p(k)$$
$$\forall k \in K, \forall n \in sp(k) : s(n) = 1 \wedge n \in p(k)$$
$$\forall k \in K : \exists (m, n) \in card(k) : m \leq |sp(k)| \leq n.$$

3.2 Evaluate Logical Dependencies

As stated in section 2.3, logical dependencies restrict possible configurations. Therefore, it is necessary to assign formal semantics to given dependencies. In Table 1, we defined the dependencies *requirement*, *dependency*, and *prohibition*.

A requirement $requires(c_1) = c_2$ between component c_1 and c_2 states selecting component c_1 leads to the selection of component c_2. This can be formalised as follows: $\forall c_1, c_2 \in C : s(c_1) = 1 \rightarrow s(c_2) = 1$.

If component c_1 depends on component c_2 ($depends(c_1) = c_2$), it is not possible that component c_2 is not selected while component c_1 is selected: $\forall c_1, c_2 \in C : s(c_2) = 0 \rightarrow s(c_1) = 0$.

Finally, prohibition of components c_1 and c_2 ($prohibits(c_1) = c_2$) permits both components being selected at the same time: $\forall c_1, c_2 \in C : (s(c_1) = 1 \rightarrow s(c_2) = 0) \wedge (s(c_2) = 1 \rightarrow s(c_1) = 0)$.

Based on the semantics of the logical dependencies, it is possible to establish valid configurations. However, as can be seen from the specification, it is also possible to establish models that are not satisfiable. For example, the logical dependencies $requires(A) = B$ and $prohibits(A) = B$ must not occur in the same model. However, satisfiability of models is not in the focus of this work. The interested reader is referred to [2] for a detailed overview about interactions between different logical dependencies.

3.3 Evaluate Temporal Dependencies

After a configuration that satisfied the given cardinalities and logical dependencies has been established, it is possible to represent the components as a procedural process. This representation can be used as input for workflow management systems that guide through a configured process. Therefore, it is necessary to arrange the components according to the given temporal dependencies.

As stated in section 2.3, we specify temporal dependencies using LTL. In the following, we first show how the temporal dependencies of Table 2 are enriched with formal semantics. Based on this semantics, we show an example configuration in its process representation in the next section.

The precedence dependency $before(c_1) = c_2$ states that in a configuration containing both components c_1 and c_2, component c_2 must be executed before component c_2. In LTL terms, this can be represented as the constraint that c_1 cannot be executed until c_2 was executed: $\forall c_1, c_2 \in C : (s(c_1) = 1 \wedge s(c_2) = 1) \rightarrow (\neg c_1 \mathcal{U} c_2)$.

The direct precedence dependency $iBefore(c_1) = c_2$ states that in a configuration containing both components c_1 and c_2, component c_2 must be executed immediately before c_1. This is an extension of the precedence dependency. In addition, it is necessary that the execution of c_1 follows immediately after the execution of c_2: $\forall c_1, c_2 \in C : (s(c_1) = 1 \wedge s(c_2) = 1) \rightarrow (\neg c_1 \mathcal{U} c_2 \wedge c_2 \rightarrow \bigcirc c_1)$.

The succession dependency $after(c_1) = c_2$ states that in every configuration containing both component c_1 and c_2, component c_2 must be executed after component c_1. In LTL terms, this can be represented as the constraint that after the execution of c_1 eventually c_2 must be executed in the future: $\forall c_1, c_2 \in C : (s(c_1) = 1 \wedge s(c_2) = 1) \rightarrow (c_1 \rightarrow \Diamond c_2)$.

Finally, direct succession $iAfter(c_1) = c_2$ implies that immediately after the execution of component c_1, component c_2 must be executed. This can be formalised similar to direct precedence. However, in this case it is not necessary that c_1 is executed before c_2 can be executed: $\forall c_1, c_2 \in C : (s(c_1) = 1 \wedge s(c_2) = 1) \rightarrow (c_1 \rightarrow \bigcirc c_2)$.

3.4 Procedural Process Transformation

With the semantics of the temporal dependencies at hand, it is possible to establish a procedural process model based on a given configuration. For comprehensibility, we show the transformation of a configured component model into a process model using the photovoltaics example. A configuration is established based on customer requirements where different ways of asking for these requirements are possible. For example, [11] shows a dialogue-driven approach to establish configurations. We support configuration decisions by our proposed hierarchical, logical, and temporal dependencies between components. In a typical example, a customer asks for a photovoltaics installation that is constructed by the service provider. In addition, the customer wants to buy the cleaning service and does not want to maintain the installation on her own. Therefore, remote maintenance is necessary. Due to hierarchical (e.g. cleaning is a child component

of maintenance) and logical dependencies (e.g. remote maintenance needs the analysis component), we have the following configuration including all necessary components.

$$Configuration = \{photovoltaics, installation, delivery, assembling, maintenance, cleaning, remote-maintenance, monitoring, analysis\}$$

Based on the hierarchic dependencies between components this configuration can be represented as a configured graph shown in Fig. 2. This graph does not contain any connectors because all components are mandatory. Since the graph should only act as a helpful visualisation, we do not formalise its constituents.

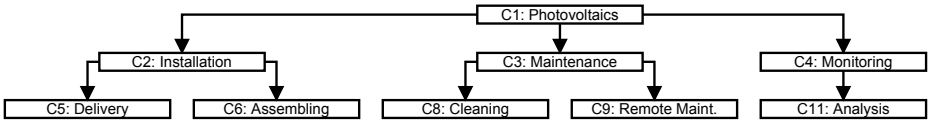


Fig. 2. Configured graph for selected components

In the following, the configured graph is transformed into a procedural process representation using Business Process Model and Notation (BPMN [12]) as the process notation. In doing so, every selected component can be represented as a (possible expandable) activity in the process model. The transformation results in a complete process model according to the logical and temporal dependencies between selected components. By now, there does not exist a complete formalisation of the transformation process. Thus, we give a step-by-step instruction.

First, a collapsed activity for the complete photovoltaics process is created (Fig. 3(a)). This is to show that we use components as refineable activities in the process model. Thus, it is possible to represent hierarchic process dependencies.

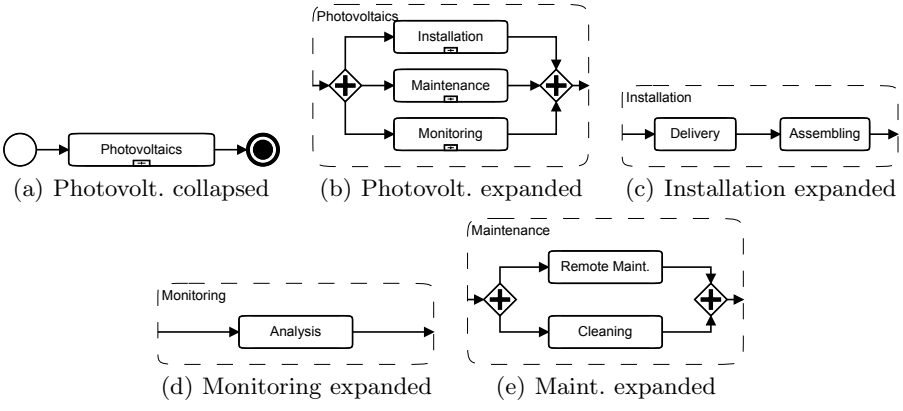
Going down one level in the configured graph, we have the components installation, maintenance, and monitoring. As stated above, all of these components are necessary. Furthermore, there are no temporal dependencies between these components. Thus, we can create an activity for each component and connect them using a parallel gateway when expanding the photovoltaics activity (Fig. 3(b)). Every components of this level is again represented as an activity that can be expanded.

When expanding the installation activity, we have to satisfy the temporal constraint $before(assembling) = delivery$. Therefore, it is necessary that the activity delivery is executed before the activity assembling, i.e. both activities must be in sequential order. Since we only have these two activities in the respective subprocess, we can directly connect both activities with each other (Fig. 3(c)).

The remaining two activities (monitoring and maintenance) are expanded in similar way. In monitoring there is only one activity analysis. Therefore, we do not have to consider any temporal constraints (Fig. 3(d)). The two maintenance

activities can be executed in parallel and are thus connected using a parallel gateway (Fig. 3(e)).

The activities in Fig. 3(c) - 3(e) cannot be expanded anymore. Therefore, the transformation is finished resulting in a complete procedural process model.



(a) Photovolt. collapsed (b) Photovolt. expanded (c) Installation expanded

(d) Monitoring expanded (e) Maint. expanded

Fig. 3. Transformation of configuration into procedural process representation

4 Related Work

In our approach we use a component model representation to allow for configuration of complex business processes. Other approaches extend existing process notations with explicit representation of variabilities. For example, Rosemann and van der Aalst extend event-driven process chains (EPCs) to so-called configurable EPCs (C-EPCs) to represent configurable reference models [13]. Therefore, they give a formal definition of C-EPCs and describe how configuration decisions effect resulting process models. Another streamline of research focuses BPMN extensions to display variability, e.g. [14,15,16]. However, a big differences between these approaches and our presented approach lies in the fact that we analyse the whole portfolio of process models of an organisation. Opposing to this, C-EPCS and BPMN extension are on the level of one specific process model.

Feature models are well-known in the domain of software engineering and another feasible approach to represent variability [17]. By establishing a distinct feature model, it is possible to extract variability from the model and to define clearly distinctable feature decision points. In doing so, existing process models can be reused without any changes. It is just necessary to group them according to included features and map the features in the feature model to the respective processes. La Rosa et al. use this approach for reference model configuration by describing variability of a domain and of a process in separate models [18].

Since we also use declarative elements in our component model, approaches for specifying declarative process models are of interest, too. Aalst and Pesic

have proposed a comprehensive overview about declarative modelling in [19]. Their work can be used as a source for additional temporal dependencies between components. Furthermore, Soffer and Yehezkel have proposed declarative modelling focusing expression of variability [20]. To broaden the view on temporal dependencies, the work of Lanz et al. can be used as an additional source [21]. They present time patterns that occur in workflow systems, e.g. lags between activities and durations of activities. It is an interesting approach to analyse how these time patterns are related with temporal dependencies.

5 Evaluation and Conclusions

In this work we presented an approach to represent business processes in a hierarchic way. The proposed component model focuses configuration of processes. Therefore, it uses hierarchically structured components that are connected by distinct nodes allowing for specifying semantics of the structure. Additionally, it is possible to assign logical and temporal dependencies between components. By comparing our approach with requirements for configurable reference modelling techniques mentioned in academic literature [13], it is possible to establish a first evaluation and future research directions.

1. Differentiate between run-time and build-time decisions. Our model uses connector nodes to represent build-time decisions, i.e. configuration points of the model that need to be decided before a model is executed. By assigning leafs of the tree with process models, it is possible to allow for run-time decisions, too. Thus, we support both decision possibilities with a clear distinction between them.
2. Support configurations regarding entire processes, functions, control flow, resources, and data. In the current state we only support configuration based on process level. Resources and data are out of focus. However, enriching component descriptions with data and resource information should be possible in future developments.
3. Differentiate between mandatory and optional decisions. Connectors can be initialised with default cardinalities. However, in the current state it is still necessary to select specific components during configurations. In future extensions of the model, default cardinalities can be enriched with the specification of default components that are selected when no explicit decision was made.
4. Differentiate between global and local decisions. Global decisions are based on specific context factors (e.g. country, domain etc.). Currently, it is not possible to map these context factors on decisions. However, an extension of the model includes so-called external variables [22]. Based on these variables it may be possible to define configuration decisions.
5. Differentiate between critical and non-critical decisions. In the current state, we support only non-critical decisions, i.e. every decisions can be re-done and can be changed over time. Thus, it is not possible to distinguish between these two decision types.

6. Depict interrelationships between configuration decisions. Due to the hierarchic representation of the component model, there is a natural configuration order, i.e. if a superior component is not selected, the child components cannot be selected, too. However, there exists no such order for logical dependencies. This is a current weakness of our component model and needs to be overcome in future research.
7. Differentiate between configuration decisions on different levels. Since we do not cover organisational details in our models, this differentiation is not contained in the model. Nonetheless, it is possible that configuration is conducted step-wise. In doing so, different levels can refine a configuration.
8. Relate variation points with additional information. Additional information are not formally defined in the model. However, the definition of connectors may be enriched with an additional information, e.g. an URL. At this URL, configuration information can be placed.
9. Guide the configuration by recommendations and guidelines. It is possible to assign key performance indicators (KPIs) to components [23]. Based on these KPIs, the productivity impacts of configuration decisions can be assessed. Organisations can further use these information to develop configuration guidelines. Additionally, the configuration process can be supported by defining recommendations for components using logical dependencies. Other research approaches promote using questionnaire-based configuration, e.g. [18]. This is possible using our approach, too. In future research we will analyse ways to establish a guided configuration.
10. Make complexity manageable. Due to the modularised, hierarchic structure of our component model, it is possible to separate process modelling from configuration. For example, executive management of an organisation uses component models on a very abstract level to decide about the overall organisation strategy. Functional departments can build on this configuration with their own, refined models.

In future, the consequences of our approach for modelling practice can be evaluated based on two approaches. First, it is possible to establish reference process models based on existing reference processes for existing domains, e.g. SAP R/3 [24]. According to the representation of these reference processes, we will analyse the complexity differences in configuring an existing reference model in comparison to the configuration using our presented approach. Second, we will analyse processes of our industry partners in more details and evaluate them according to their configuration potential.

Since the first-order and linear temporal logic formalisation is not easy to use (especially for non-professionals) we have developed a tool conforming to the notation. The practical applicability of this tool is shown in [25]. However, in its current state the tool cannot transform component into process models. To further enhance practical applicability, we have to analyse how existing process models can be reused (extraction). This is necessary, since organisations often

own process repositories consisting of hundreds or even thousands of models [26]. A valid starting point for extraction is to identify mappings between workflow patterns according to [27] and specific component hierarchies.

References

1. Böttcher, M., Klingner, S.: The basics and applications of service modeling. In: SRII Global Conference 2011 (2011)
2. Becker, M., Klingner, S.: Formale Modellierung von Komponenten und Abhängigkeiten zur Konfiguration von Product-Service-Systems. To Appear in: Diensleistungsmodellierung 2012 (March 2012)
3. Pohl, K., Böckle, G., Linden, F., Lauenroth, K., Pohl, K.: Principles of Variability. In: Software Product Line Engineering, pp. 57–88. Springer, Heidelberg (2005)
4. Becker, M.: Formales Metamodell für Dienstleistungskomponenten. Technical report, Universität Leipzig (2011)
5. Heiskala, M., Paloheimo, K.S., Tiihonen, J.: Mass Customization of Services: Benefits and Challenges of Configurable Services. In: Frontiers of e-Business Research (FeBR 2005), Tampere, Finland, pp. 206–221 (September 2005)
6. Thum, T., Batory, D., Kastner, C.: Reasoning about edits to feature models. In: Proceedings of the 31st International Conference on Software Engineering, ICSE 2009, pp. 254–264. IEEE Computer Society, Washington, DC (2009)
7. van den Broek, P., Galvão, I., Noppen, J.: Elimination of constraints from feature trees. In: Thiel, S., Pohl, K. (eds.) Proceedings of 12th International Conference on Software Product Lines, SPLC 2008, Limerick, Ireland, September 8-12, vol. 2(Workshops), pp. 227–232 (2008)
8. Gil, Y., Kremer-Davidson, S., Maman, I.: Sans Constraints? Feature Diagrams vs. Feature Models. In: Bosch, J., Lee, J. (eds.) SPLC 2010. LNCS, vol. 6287, pp. 271–285. Springer, Heidelberg (2010)
9. van der Aalst, W., Pesic, M.: Specifying, Discovering, and Monitoring Service Flows: Making Web Services Process-Aware. BPM Center Report BPM-06-09, BPMcenter.org (2006)
10. Sistla, A.P., Clarke, E.M.: The complexity of propositional linear temporal logics. J. ACM 32, 733–749 (1985)
11. Tiihonen, J., Soininen, T., Niemelä, Sulonen, R.: A practical tool for mass-customising configurable products. In: Proceedings of the 14th International Conference on Engineering Design, pp. 1290–1299 (2003)
12. Grosskopf, A., Decker, G., Weske, M.: The Process: Business Process Modeling Using BPMN, 1 edn. Meghan Kiffer Pr, Tampa (March 2009)
13. Rosemann, M., van der Aalst, W.: A configurable reference modelling language. Information Systems 32(1), 1–23 (2007)
14. Montero, I., Pena, J., Ruiz-Cortes, A.: Representing runtime variability in business-driven development systems. In: International Conference on Commercial-off-the-Shelf (COTS)-Based Software Systems, vol. 1, p. 241 (2008)
15. Rogge-Solti, A., Kunze, M., Awad, A., Weske, M.: Business process configuration wizard and consistency checker for bpmn 2.0. In: Halpin, T., Nurcan, S., Krogstie, J., Soffer, P., Proper, E., Schmidt, R., Bider, I., Aalst, W., Mylopoulos, J., Rosemann, M., Shaw, M.J., Szyperski, C. (eds.) BPMDS 2011 and EMMSAD 2011. LNBIP, vol. 81, pp. 231–245. Springer, Heidelberg (2011)

16. Santos, E., Pimentel, J., Castro, J., Sánchez, J., Pastor, O.: Configuring the Variability of Business Process Models Using Non-Functional Requirements. In: Bider, I., Halpin, T., Krogstie, J., Nurcan, S., Proper, E., Schmidt, R., Ukor, R. (eds.) BPMDS 2010 and EMMSAD 2010. LNBIP, vol. 50, pp. 274–286. Springer, Heidelberg (2010)
17. Batory, D.: Feature Models, Grammars, and Propositional Formulas. In: Obbink, H., Pohl, K. (eds.) SPLC 2005. LNCS, vol. 3714, pp. 7–20. Springer, Heidelberg (2005)
18. La Rosa, M., Gottschalk, F., Dumas, M., van der Aalst, W.M.P.: Linking Domain Models and Process Models for Reference Model Configuration. In: ter Hofstede, A.H.M., Benatallah, B., Paik, H.-Y. (eds.) BPM Workshops 2007. LNCS, vol. 4928, pp. 417–430. Springer, Heidelberg (2008)
19. 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)
20. Soffer, P., Yehezkel, T.: A State-Based Context-Aware Declarative Process Model. 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. 148–162. Springer, Heidelberg (2011)
21. Lanz, A., Weber, B., Reichert, M.: Workflow Time Patterns for Process-Aware Information Systems. In: Bider, I., Halpin, T., Krogstie, J., Nurcan, S., Proper, E., Schmidt, R., Ukor, R. (eds.) BPMDS 2010 and EMMSAD 2010. LNBIP, vol. 50, pp. 94–107. Springer, Heidelberg (2010)
22. Becker, M., Klingner, S., Böttcher, M.: Configuring services regarding service environment and productivity indicators. In: 2011 Federated Conference on Computer Science and Information Systems (FedCSIS), pp. 505–512 (September 2011)
23. Böttcher, M., Klingner, S.: Providing a Method for Composing Modular B2B-Services. Journal of Business and Industrial Marketing 26(5), 320–331 (2011)
24. Curran, T., Keller, G., Ladd, A.: SAP R/3 business blueprint: understanding the business process reference model. Prentice-Hall, Inc., Upper Saddle River (1998)
25. Klingner, S., Böttcher, M., Becker, M., Döhler, A.: Managing complex service portfolios. In: Ganz, W., Kicherer, F., Schletz, A. (eds.) RESER 2011 Productivity of Services NextGen - Beyond Output/Input (September 2011)
26. Dumas, M., García-Bañuelos, L., Dijkman, R.: Similarity Search of Business Process Models. IEEE Data Engineering Bulletin 32(3), 23–28 (2009)
27. van der Aalst, W., ter Hofstede, A., Kiepuszewski, B., Barros, A.: Workflow Patterns. Distributed and Parallel Databases 14, 5–51 (2003)