Declarative Enhancement Framework for Business Processes *

Heerko Groefsema, Pavel Bulanov, and Marco Aiello

Distributed Systems Group, Johann Bernoulli Institute, University of Groningen,
Nijenborgh 9, 9747 AG Groningen, The Netherlands
{h.groefsema, p.bulanov, m.aiello}@rug.nl
http://www.cs.rug.nl/ds/

Abstract. While Business Process Management (BPM) was designed to support rigid production processes, nowadays it is also at the core of more flexible business applications and has established itself firmly in the service world. Such a shift calls for new techniques. In this paper, we introduce a variability framework for BPM which utilizes temporal logic formalisms to represent the essence of a process, leaving other choices open for later customization or adaption. The goal is to solve two major issues of BPM: enhancing reusability and flexibility. Furthermore, by enriching the process modelling environment with graphical elements, the complications of temporal logic are hidden from the user.

Keywords: BPM, Variability, Temporal Logic, e-Government

1 Introduction

The world of Business Process Management (BPM) has gone through some major changes [4] due, among other things, to the advent of Web services and Service-orientation; providing opportunities as well as challenges [2]. Variability is an abstraction and management method that addresses a number of the open issues. In the domain of software engineering, variability refers to the possibility of changes in software products and models [13]. When this is introduced to the BPM domain, it indicates that parts of a business process remain either open to change, or not fully defined, in order to support different versions of the same process depending on the intended use or execution context, see for instance our survey [3]. Since BPM is moving into more fields of business and rely on autonomous remote building blocks, a need for flexible processes has arisen. Today, when a number of closely related processes are in existence, they are either described in different process models or in one large model using intricate branching routes, resulting in redundancy issues in case of the former and maintainability and readability issues in the case of the latter [14, 3]. When applied to process models, variability introduces solutions to both issues by offering support for reusability and flexibility.

* The research is supported by the NWO SaS-LeG project, http://www.sas-leg.net, contract No. 638.001.207.

Considering the two features introduced by variability, we immediately notice how the two tend to coincide with design– and run–time aspects. The reusability attribute is usually introduced at design–time, and the flexibility attribute at run–time. Generally two approaches to variability are considered, imperative and declarative ones [12]. While imperative approaches focus on how a task is performed, declarative approaches focus on what tasks are performed. When mapping these to the two areas where variability would operate, design– and run–time, we notice four possible directions regarding variability in BPM. Most research currently focuses on the areas of imperative/design–time and declarative/run–time, while in this paper we shall focus on an approach which is able to capture both imperative and declarative methods at design–time [3]. The common problem with the frameworks of the imperative/design–time area is that in many cases they introduce unnecessary restrictions for process designers [1]. The source of such restrictions lies in the fact that imperative approaches require all variations from the main process to be predefined, limiting variations to only those added explicitly. On the other hand, declarative run–time frameworks lie on the far side of the semantic gap between the traditional and well–understood way of imperative process specification and the unintuitive way of declarative specification. We intend to solve these issues through a design–time declarative framework, in which the principles of process designing are similar to the ones of traditional imperative–based process modelling. This is achieved via introducing a set of visual modelling elements, which are internally transposed into a set of declarative constraints.

In this paper, we start by introducing the Process Variability - Declarative'n'Imperative (PVDI) framework which enhances process models with variability management through the introduction of temporal logics which capture the essence of a process. In doing so, we provide both the BPM and service composition domains with a formal way to model processes such that they can be used as a template for either the modelling of process variants or automatic service composition. In addition, by enriching the traditional process modelling environment with graphical elements, the complications of the underlying temporal logic is hidden from the user. We then show the expressive power of these graphical elements by utilizing them on an e-Government related case–study. Then, we evaluate the strengths and weaknesses of the framework by looking at the requirements for variability management in service-oriented systems introduced in [3]. Finally, we conclude with related work, and some final remarks.

2 The PVDI Framework

A PVDI *process* is defined as a directed graph and can serve as a frame for a modal logic of processes, including computational tree logic$^+$(CTL^+)[6]. Using CTL^+, we can introduce *constraints* for processes, which allow us to capture the basic meaning of a process in such a way that we can control changes within the process while keeping its essence, its intended use, intact as long as none of these

constraints are violated. It is then possible to allow anyone to design a variant from such a template process without compromising its intended use.

Definition 1 (Process). *A process P is a tuple $\langle A, G, T\rangle$ where:*

- *A is a finite set of activities, with selected start $\odot$ and final $\otimes$ activities;*
- *$G = G_a \cup G_o \cup G_x$ is a set of gateways, consisting of and, or, and xor gates as defined by BPMN, respectively;*
- *$S = A \cup G$ is a set of states;*
- *$T = T_a \cup T_g$, where:*
- *$T_a : (A \backslash \{\otimes\}) \to S$ is a finite set of transitions, which assign a next state for each activity;*
- *$T_g : G \to 2^S$ is a finite set of transitions, which assign a nonempty set of next states for each gateway.*

In order to use a process as a model, we introduce a set of variables and a valuation function. We use the so–called **natural** valuation, that is, for each state (i.e., for each activity or gateway) we introduce its dedicated variable, and this variable is valuated to TRUE on this state only. Additionally, under the natural valuation we can use the same letter to represent both activity and its corresponding variable.

Definition 2 (Constraint). *A* **constraint** *over the process P is a computation tree logic$^+$ (CTL^+) formula. A constraint is* **valid** *for a process P iff it is valuated to TRUE in each state of the process under the natural valuation. More formally, let ϕ be a constraint, $\mathcal{M}$ be a model built on the process P using the natural valuation, and S be the set of states of the process P. Then ϕ is valid iff $\mathcal{M}, x, \phi \models TRUE \ \forall x \in S$*

Notice how, now that we introduced constraints, a template process can be defined without strictly defining the precise structure of all activities and their respective ordering. The mapping of T can therefore be a partial one. As a result, a template may range from being a list of tasks to being a fully specified process. On the other hand, templates are enriched with a set of constraints, which outline the general shape of a process or set of processes.

Definition 3 (Template). *A template T is a tuple $\langle A, G, T, \Phi\rangle$ where:*

- *A, G, S, and T are from Definition 1;*
- *$T_a : (A \backslash \{\otimes\}) \to S$ is a finite set of transitions, which assign a next state for* **some** *activities;*
- *$T_g : G \to 2^S$ is a finite set of transitions, which assign a nonempty set of next states for* **some** *gateways.*
- *Φ is a finite set of constraints.*

In order to facilitate template design we introduce a number of graphic–elements for the design process. Constraints as embedded within templates are then generated from these graphical–elements. For every element contained in the template,

we generate one or more CTL^+ formulas. Of course, this process differs greatly per element and even per situation. Some of the simpler elements directly resemble simple CTL^+ formulas, whereas other more complex structures resemble a set of CTL^+ formulas . A potentially large number of graphic–elements can be considered for the template design, ranging from simple flows to complex groupings of tasks. We discuss only those elements needed in order to introduce the wide variety of options available trough PVDI.

2.1 Flow Constraints

Flow Constraints state that all elements from one set are followed by at least one element from another set in either a single or all paths. With a path being a series of consecutive transitions.

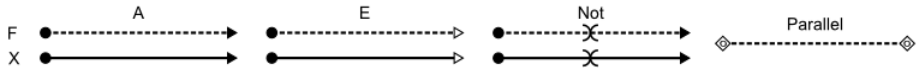


Fig. 1. Flow elements.

Definition 4 (Flow Constraint). *A Flow Constraint F is a tuple $\langle s, t, \Omega, \Pi, N\rangle$ where:*

- *s is a finite set of process elements;*
- *t is a finite set of process elements; $s \bigcap t = \emptyset$;*
- *$\Omega \in \{A, E\}$ is a state quantifier from CTL^+;*
- *$\Pi \in \{X, F, G\}$ is a path quantifier from CTL^+;*
- *$N \in \{TRUE, FALSE\}$ being a negation.*

The graphical–elements related to Flow Constraints are shown in Figure 1. These graphical–elements describe flow relations over two dimensions; path and distance. The rows of Figure 1 relate to the temporal dimensions; F (Finally) and X (neXt), which require the linked elements to either follow each other eventually or immediately. The first two columns relate to the paths; E (there Exists a path) and A (for All paths), which require the linked elements to follow each other in either a path or all paths respectively. The third column represents a negation of two of these flows. These flows are related to simple CTL^+ formulas of the form $\mathcal{M}, p \models AXq$ (At p, in All paths, the neXt task is q), $\mathcal{M}, p \models EFq$ (At p, there Exists a path, where Finally a task is q), and their negations.

The CTL^+ constraints are generated from the graphical–elements according to the steps below. Remember that we reuse the same symbol to represent both the state and the variable representing that state.

1. Let $s_1, \ldots s_n$ be elements of *source*, and $t_1, \ldots t_m$ be elements of *target*.
2. If $N = TRUE$, then create a formula $(s_1 \vee s_2 \vee \ldots s_n) \Rightarrow \Omega\Pi(t_1 \vee t_2 \vee \ldots t_m)$.
3. If $N = FALSE$, then create a formula $(s_1 \vee s_2 \vee \ldots s_n) \Rightarrow \Omega\Pi\neg(t_1 \vee t_2 \vee \ldots t_m)$.
4. If $N = FALSE$ and $\Pi = F$, then create a formula $(s_1 \vee s_2 \vee \ldots s_n) \Rightarrow \Omega G\neg(t_1 \vee t_2 \vee \ldots t_m)$.

2.2 Parallel Constraints

Parallel Constraints (Figure 1), enforce that two sets of states do not appear in the same path. Meaning that any series of consecutive transitions taken from any state in either set may never lead to any element from the other set.

Definition 5 (Parallel Constraint). *A Parallel Constraint P is a tuple $\langle S, T \rangle$ where:*

- *S and T are sets of states; $S \bigcap T = \emptyset$;*

The CTL^+ constraints are generated from the graphical–elements according to the steps below. This construction enforces that all states from each set or branch, S and T, can never be followed by any state from the other set. Further constraints between states in the branches S and T, and the specification of a specific preceding gate should be added through other means, i.e. flow constraints.

1. Let $s_1, \ldots s_n$ be elements of S, and $t_1, \ldots t_m$ be elements of T.
2. Create a formula $(s_1 \vee s_2 \vee \ldots s_n) \Rightarrow AG\neg(t_1 \vee t_2 \vee \ldots t_m)$
3. Create a formula $(t_1 \vee t_2 \vee \ldots t_m) \Rightarrow AG\neg(s_1 \vee s_2 \vee \ldots s_n)$

2.3 Frozen Groups

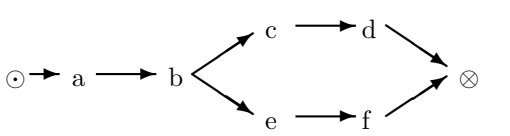
Frozen Groups are sub–processes which cannot be modified. Such a restriction is achieved by generating a set of CTL^+ formulas which constrain all elements inside of a frozen group. Figure 2 includes an example of such a group.

Definition 6 (Frozen group). *A frozen group is a pair $\langle P, M \rangle$, where P is a process (Definition 1), and $M \subseteq P_A$ is a set of mandatory activities.*

The group itself is a process but can be part of a bigger process or template, which is why we refer to groups as sub–processes. Since this element is not limited to one set of simple sources and targets like the previous ones, its transformation to CTL^+ is more complicated, and yields a set of CTL^+ formulas instead of a single one. A Frozen Group will be constrained in such a way that all paths, being a series of consecutive transitions, within it, must be kept intact. The CTL^+ constraints are generated from the graphical–elements (Figure **??**) according to the steps below. be no changes in its contents.

1. For each state a let the chain of states $P = \langle a_1, \ldots a_k \rangle$ be a path between a and $\otimes$. If the path P is not empty, then do the following steps:
2. If the path P is empty (i.e., the next step is the final one), then create a CTL^+ formula of type $a \Rightarrow AX\otimes$.
3. If the path P is the only path from a to $\otimes$, then create a CTL^+ formula of type $a \Rightarrow A((a_1 \vee a_2 \vee \ldots \vee a_k)U\otimes)$. A and U are quantifiers of CTL^+.
4. If there are several paths which lead from a to $\otimes$, let say paths $P_1, \ldots P_t$ lead from a to $\otimes$. Then, the formula of step 3 becomes more complicated: $a \Rightarrow A[P_1^T \vee \ldots \vee P_t^T]$, where $P_i^T = (a_1^i \vee a_2^i \vee \ldots \vee a_k^i)U\otimes$, with $a_1^i \ldots a_k^i$ being the steps of the path P_i.

Example 1 (Frozen Group).



$a \Rightarrow A[(b \vee c \vee d)U\otimes] \vee$
$\quad [(b \vee e \vee f)U\otimes]$
$b \Rightarrow A[(c \vee d)U\otimes] \vee [(e \vee f)U\otimes]$
$c \Rightarrow A(dU\otimes)$
$e \Rightarrow A(fU\otimes)$
$d \Rightarrow AX\otimes$
$f \Rightarrow AX\otimes$

In this example, the process illustrated above is encoded as a set of CTL^+ formulas. To do that, we take each step one by one and generate a formula according to the algorithm presented above. The first two formulas are the most complicated, because there are two possible paths from a to $\otimes$ (and from b to $\otimes$ as well). Therefore, according to the step 4 of the algorithm, the formula splits into two pieces, one per each path. In the case of activity b, one path contains activities c and d, and the other one contains e and f.

2.4 Semi–frozen Group

Semi–frozen groups are Frozen Groups with less strict constrains, allowing for removal or replacement, addition, or moving of activities. The advantage of this group representation is that for example any activity inside a frozen group can be made optional and can therefore be removed during the customization process.

Optional activities will not affect the consistency of the group, since the CTL^+ formulas remain valid as long as at least no new activity is added into the group. Actually, if an activity a is removed, then all formulas of kind $a \Rightarrow \ldots$ become automatically valid, and formulas of kind $b \Rightarrow aU\otimes$ are valid as long as there is either activity a or nothing between b and $\otimes$. The same is true for more complicated cases like $b \Rightarrow a \vee \ldots U\otimes$. In other words, any activity can be removed from a frozen group but not replaced by another one.

Weaken a link between two states thus allowing to insert a new activity into a specific place(s) in a group. To do that, we have to modify the base algorithm.

1. Let the chain of states $P = \langle a_1, \ldots a_k \rangle$ be a path between a and $\otimes$. For example, the link between a_i and a_{i+1} is "weak". Then the appropriate CTL^+ formula is $a \Rightarrow A[(a_1 \vee \ldots a_i)UAFA[(a_{i+1} \vee \ldots a_n)U\otimes]]$.
2. If there are several "weak" links in a path, for example, links $a_i \rightarrow a_{i+1}$ and $a_j \rightarrow a_{j+1}, i < j$ are weak. In this case, build a formula $a_j \Rightarrow \phi$ according to p.1, and the final formula is $a \Rightarrow A[(a_1 \vee \ldots a_i)UA[(a_{i+1} \vee \ldots a_j)U\phi]]$, where ϕ is retrieved in the previous step.
3. The same recursive rule applies in the case of three or more "weak" links.

Making an activity floating thus allowing to swap two activities or drag an activity into another place in the group. The algorithm is as follows:

1. Create the set of constraints for the group as described above;
2. To make an activity a floating, remove all constraints of kind $a \Rightarrow \phi$.

Each move of an activity can be split into two atomic operations: (i) remove the activity (ii) insert this activity into another place. As it has already been shown above, no special correction needed in order to remove the activity. The next step, however, is only possible when there is a "weak" link in the process – otherwise we can only put the activity back into its original place.

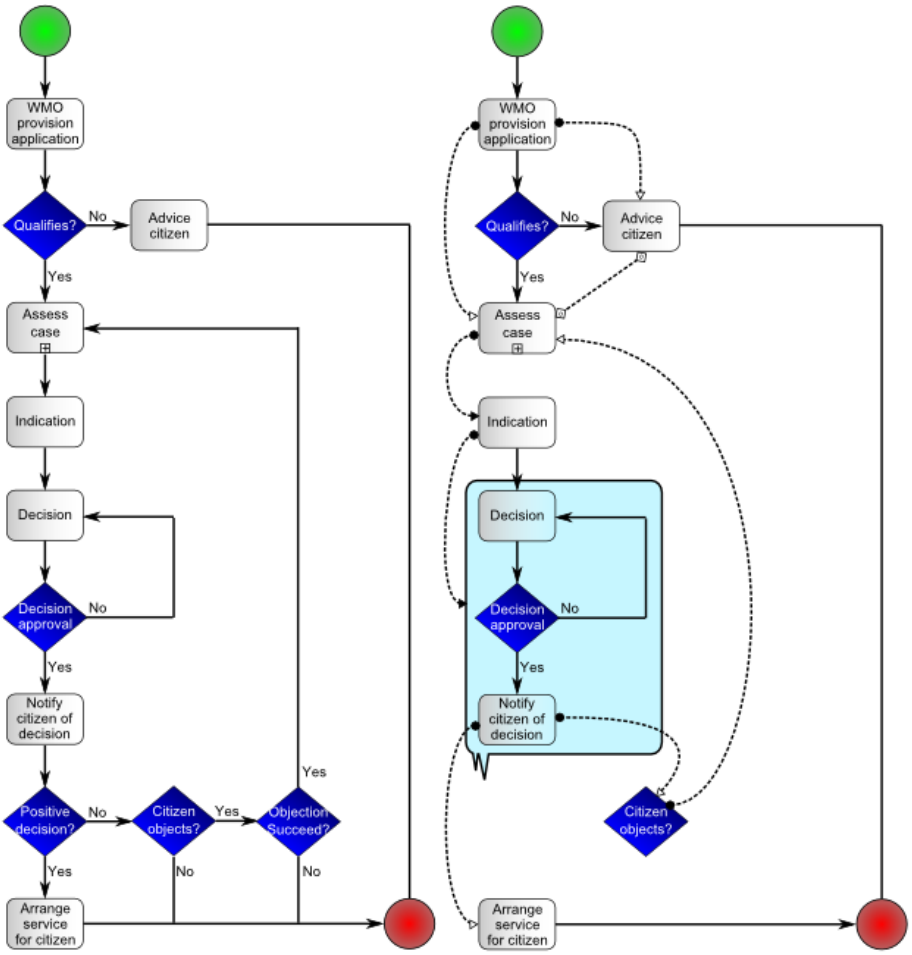


Fig. 2. Simplified WMO process (Left) and template (Right).

3 Case–Study: Variability in local eGovernment

The Netherlands consists of 418 municipalities which all differ greatly. Because of this, each municipality is allowed to operate independently according to their local requirements. However, all the municipalities have to provide the same services and execute the same laws. An example of such a law which is heavily subjected to local needs is the WMO (Wet maatschappelijke ondersteuning, Social Support Act, 2006), a law providing needing citizens with support ranging from wheelchairs, help at home, home improvement and homeless sheltering. Figure 2 illustrates a simplified version of a WMO process found at one of the Dutch municipalities of the Northern region of the Netherlands (Left), and an example of how one transforms this process into a template usable by all municipalities (Right). Variant processes can then be obtained via customization of the template process. The flexibility of a process, bounded with such constraints, can vary from zero (a frozen block over the whole process) to unlimited, when there are no constraints at all. On examination of the figure, we notice how constraints are not evaluated for templates but only for processes resulting from templates. Using a frozen group we restrict the decision making process, which because of this must be kept intact at all times. The rest of the template is captured using simple flow constraints and one parallel constraint. Therefore, extra activities could be easily included at most places except the frozen group, and certain parts could be moved around without affecting the correctness of the process.

4 Related Work

Existing tools and frameworks for variability management in BPM concentrate on a single view on variability. Most focus either on imperative design–time or on declarative run–time solutions, while we combine imperative and declarative techniques. In addition, most research disregards services entirely and focuses solely on the BPM aspects. One framework which does look at services specifically is the Variability extension to Business Process Execution Language (VxBPEL) [15] that we proposed previously. This BPEL extension introduces a number of new keywords allowing for the inclusion of variation points, variations, and realization relations into BPEL. Other imperative frameworks focus solely on BPM, most notably ADEPT [5], Process Variants by Options(Provop) [8], and configurable workflow models [7]. These imperative frameworks however require that all variability options must be included into the template process directly, leading to maintainability and readability issues. On the other hand, our framework does not focus on what can be done, but on what should be done, and leaves any other options open; resulting in a much higher degree of flexibility. Declarative frameworks focus mostly on run–time solutions to flexibility issues, of which most notable are the DECLARE framework [10], and Business Process Constraint Network (BPCN) and Process Variant Repository(PVR) [9, 11]. Our framework on the contrary hides this complexity by the introduction of simple graphical elements which can be directly incorporated into business process models.

	Requirement	Support
	Structural variations	
(a)	*Insert Process Fragment*	Achieved via introducing a "weak" link in a frozen group.
(b)	*Delete Process Fragment*	Achieved via making an activity optional.
(c)	*Move Process Fragment*	Achieved via an floating activity at frozen groups.
(d)	*Replace Process Fragment*	Achieved through a combination of an optional activity and a weak link.
(e)	*Swap Process Fragment*	Special case of moving a process fragment, therefore it is also supported.
	Constraint expressions	
(a)	*Mandatory selection*	Achieved with PVDI through a flow constraint emerging from $\odot$ and targeting the activity.
(b)	*Prohibitive selection*	Achieved with PVDI through a negated flow constraint emerging from $\odot$ and targeting the prohibited activity.
(h)	*Mandatory execution*	Achieved with PVDI through a flow constraint of the type "for All paths" emerging from $\odot$.
(i)	*Order of execution*	Achieved through flow constraints.
(j)	*Parallel execution*	Achieved through a combination of flow constraints emerging from a gate and a parallel constraint.
(k)	*Exclusive execution*	Achieved through a combination of flow constraints emerging from a gate and a parallel constraint.

Table 1. Evaluation on Requirements

5 Conclusion

We have shown how a variability framework for process modelling adds a large amount of functionality to both the area of BPM, as well as service composition. By enriching the process modelling environment with graphical elements, we provided an easy way to hide the complications of temporal logic from the end user. Using a case–study from the area of e–Government, we then explained how the high amount of reusability and flexibility enriches templates in such a way that variants become easily maintainable and templates easily readable.

In order to evaluate the flexibility of PVDI, we consider the expressive power requirements proposed in [3]. Two types are discussed: structural variations related to imperative techniques, and constraint expressions related to declarative techniques. In Table 1 the requirements which are directly supported by our framework are enlisted along with the description of the support. Due to the approach taken with PVDI, both declarative and imperative techniques are being considered. In cases of imperative techniques we consider a semi–frozen block, which keeps the process structure intact while allowing some modification.

Many items are open for further investigation, among which the support of data flows, dependencies between constraints, template publication, and automated composition from templates as a constraint satisfaction problem.

References

1. van der Aalst, W., Hofstede, A.H.M.T., Weske, M.: Business Process Management: A Survey. In: Int. Conference on Business Process Management. pp. 1–12 (2003)
2. van der Aalst, W., Jablonski, S.: Dealing with workflow change: Identification of issues and solutions. International Journal of Computer Systems, Science, and Engineering 15(5), 267–276 (2000)
3. Aiello, M., Bulanov, P., Groefsema, H.: Requirements and tools for variability management. In: IEEE workshop on Requirement Engineering for Services (REFS 2010) at IEEE COMPSAC (2010)
4. Bandara, W., Indulska, M., Sadiq, S., Chong, S.: Major issues in business process management: an expert perspective. In: European Conference on Information Systems (ECIS) (2007)
5. Dadam, P., Reichert, M.: The adept project: a decade of research and development for robust and flexible process support. Computer Science - R&D 23(2), 81–97 (2009)
6. Emerson, E.A., Halpern, J.Y.: Decision procedures and expressiveness in the temporal logic of branching time. In: Proceedings of the fourteenth annual ACM symposium on Theory of computing. pp. 169–180 (1982)
7. Gottschalk, F., van der Aalst, W.M.P., Jansen-Vullers, M.H., Rosa, M.L.: Configurable workflow models. Int. J. Cooperative Inf. Syst. 17(2), 177–221 (2008)
8. Hallerbach, A., Bauer, T., Reichert, M.: Managing process variants in the process life cycle. In: ICEIS (3-2). pp. 154–161 (2008)
9. Lu, R., Sadiq, S., Governatori, G.: On managing business processes variants. Data Knowl. Eng. 68(7), 642–664 (2009)
10. Pesic, M., Schonenberg, M.H., Sidorova, N., van der Aalst, W.M.P.: Constraint-based workflow models: Change made easy. In: OTM Conferences (1). pp. 77–94 (2007)
11. Sadiq, S.W., Orlowska, M.E., Sadiq, W.: Specification and validation of process constraints for flexible workflows. Inf. Syst. 30(5), 349–378 (2005)
12. Schonenberg, H., Mans, R., Russell, N., Mulyar, N., van der Aalst, W.M.P.: Process flexibility: A survey of contemporary approaches. In: Dietz, J.L.G., Albani, A., Barjis, J. (eds.) CIAO! / EOMAS. LNBIP, vol. 10, pp. 16–30. Springer (2008)
13. Sinnema, M., Deelstra, S., Hoekstra, P.: The COVAMOF Derivation Process. In: Int. Conf. On Software Reuse. vol. LNCS, pp. 101–114. Springer (2006)
14. Sun, C., Rossing, R., Sinnema, M., Bulanov, P., Aiello, M.: Modelling and managing the variability of web service-based systems. Journal of Systems and Software, Elsevier 83, 502–516 (2010)
15. Sun, C., Aiello, M.: Towards variable service compositions using VxBPEL. In: Int. Conf. On Software Reuse. vol. LNCS 5030, pp. 257–261. Springer (2008)