Imperative versus Declarative Process Variability: Why Choose?

Heerko Groefsema, Pavel Bulanov, and Marco Aiello*

JBI 2012-12-6

January 11, 2013

Abstract

Variability is a powerful abstraction in software engineering that allows managing product lines and business processes requiring great deals of change, customization and adaptation. In the field of Business Process Management (BPM) the increasing deployment of workflow engines having to handle an increasing number of instances has prompted for the strong need for variability techniques. The idea is that parts of a business process remain either open to change, or not fully defined, in order to support several versions of the same process depending on the intended use or execution context. The goal is to support two major challenges for BPM: re-usability and flexibility. Existing approaches are broadly categorized as Imperative or Declarative. We propose Process Variability through Declarative and Imperative techniques (PVDI), a variability framework which utilizes temporal logic to represent the basic structure of a process, leaving other choices open for later customization and adaptation. We show how both approaches to variability excel for different aspects of the modeling and we highlight PVDI's ability to take the best of both worlds. Furthermore, by enriching the process modeling environment with graphical elements, the complications of temporal logic are hidden from the user. To show the practical viability of PVDI, we present tooling supporting the full PVDI lifecycle and test its feasibility in the form of a performance evaluation.

*H. Groefsema, P. Bulanov, and M. Aiello are with the The Johann Bernoulli instituut for Mathematics and Computer Science, University of Groningen, Nijenborgh 9, 9747 AG Groningen.

1 Introduction

Business Process Management (BPM) is evolving rapidly due to new requirements having to do with mass customization, need for adapting to varying business and execution contexts, and the wider availability of service-based infrastructures. Where BPM supported local user–specific rigid and repetitive units of work in the past, now it often has to support loosely–coupled processes in cloud-based environments with many users possibly having highly variable requirements. With every evolution new opportunities as well as challenges arise [1]. *Variability* is an abstraction and management method that addresses a number of the related issues. In the domain of software engineering, variability refers to the possibility of changes in software products and models [2]. When introduced to the domain of BPM, it indicates that parts of a business process remain either open to change, or not fully defined, in order to support several versions of the same process depending on the intended use or execution context [3]. Currently, when multiple similar business processes are required, they either exist as one large process definition using intricate branching descriptions or in multiple separate process definitions. This makes readability and maintainability a major problem in case of processes with intricate branching routes, or creates redundancy issues in case of multiple separate process definitions [3, 4]. We propose a novel variability approach as a solution. By introducing variability to BPM in a new way, we offer support for both re-usability and flexibility, ameliorating the readability, maintainability, and redundancy issues. Re-usability and flexibility are both directly supported by the fact that variability allows change within business processes. Multiple similar but different process instances may be based upon a single re-usable process by applying several changes as allowed by the variability, and may then be adapted at run-time due to this same flexible nature.

When considering variability in the domain of BPM, two distinct approaches exist: imperative and declarative variability [5]. The first uses the typical imperative specification of business processes and offers variability through sets of specifically allowed changes at prescribed points in the business process, called *variation points*. The second introduces constraints to specify what change is disallowed. Therefore, any change not disallowed through such constraints is allowed. Variability is offered to the BPM domain at two stages of the BPM life-cycle: *design–* and *run–time*. At design–time, variability focuses primarily on enabling re-usability of the business processes, whereas at run–time it aims at allowing flexibility during process execution. As a result, four different areas towards BPM variability can be identified: process re-usability or flexibility using either imperative or declarative techniques. Of these four areas, most research has focus on re-usability using imperative techniques or flexibility using declarative techniques [3]. Re-usability using imperative techniques however presents a number of drawbacks. First, all variability must be specifically allowed within the process. Although at first sight this might not seem like a drawback, the result implies that all possible variability must be known explicitly at design–time. Obviously, when dealing with processes which include a large range of variability, several options will be overlooked, or worse, combinations which should be prevented are unnoticed and allowed in the final process. On the other hand, declarative variability lies 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 where temporal logic formulas are used to define relations between the different activities.

We propose a variability approach to re-usability and flexibility using a declarative process definition while maintaining the traditional and well–understood way of imperative process specification through the introduction of graphical design elements which are internally translated into sets of temporal logic constraints. Because we use these graphical design elements instead of the constraints themselves, we do not only support declarative variability techniques, but imperative ones as well. We present the variability framework using a case–study from the Dutch local eGovernment, and developed tool support in order to validate practicality of the approach.

Here, we expand upon Process Variability though Declarative and Imperative techniques (PVDI) as proposed in our conference paper [6]. In Section 2, we present a case–study where we discuss variability in the context of the Dutch local eGovernment. Section 3 introduces PVDI by presenting how it affects the BPM life-cycle and by defining all of its aspects. In Section 4, we present how PVDI approaches process variability through template design using both declarative and imperative constraint techniques. Section 5 introduces constraint relations, which allow relations between constraints. We then describe how PVDI supports process healthiness conditions such as the absence of dead–ends and the reachability of elements within the business process in Section 6. We describe the variant validation process, including the model conversion, the validation algorithm, and a performance test, in Section 7. Section 8 describes how the framework guides the design of variants through a room reservation example. The expressive power of the framework is evaluated in Section 9 via a careful definition and comparison of both the properties and the complexity of the declarative and imperative approaches to variability. Finally, we discuss related work in Section 10 before drawing conclusions about our proposal in Section 11.

2 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 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 1 illustrates a simplified version of the general WMO process. The simplified version is based on the commonalities between processes obtained through interviews with seven different municipalities located in the Northern area of the Netherlands [7, 8]. Municipalities interviewed range in size, population, and income, and also differed in being urban or rural areas. The process starts with an application procedure which determines if the request made by the citizen falls under the WMO law. If this would not be the case, the citizen must be advised by the municipality employee towards his next steps. In case that the request made by the citizen does fall under the WMO law, the application is added to the system, an indication is written, and a decision whether to approve the requested support, in what form, and partially or fully, is made based on the indication. The decision is then checked by a colleague, and reported back to the citizen. After which the support is arranged for the citizen in case of a positive decision, or the citizen may object against a negative decision.

Within the general WMO process as illustrated in Figure 1 a large quantity of variability was discovered through careful examination of the differences in the WMO processes found at the seven municipalities [7, 8]. Most notable was the variability concerning the information gathering activities

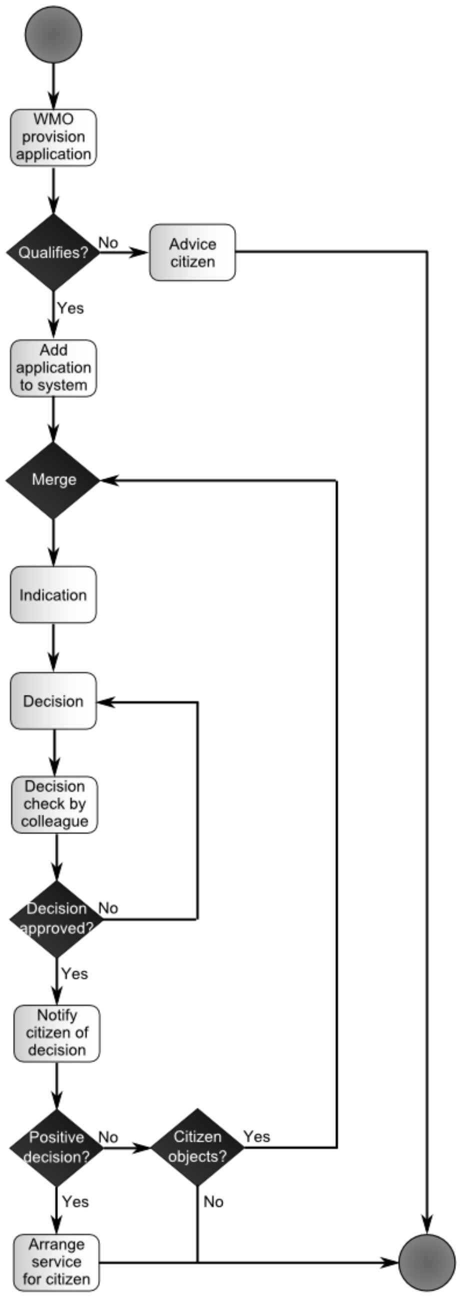

WMO provision application
Qualifies?
No
Advice citizen
Yes
Add application to system
Merge
Indication
Decision
Decision check by colleague
Decision approved?
No
Yes
Notify citizen of decision
Positive decision?
No
Citizen objects?
Yes
Yes
No
Arrange service for citizen

Figure 1: Simplified WMO process.

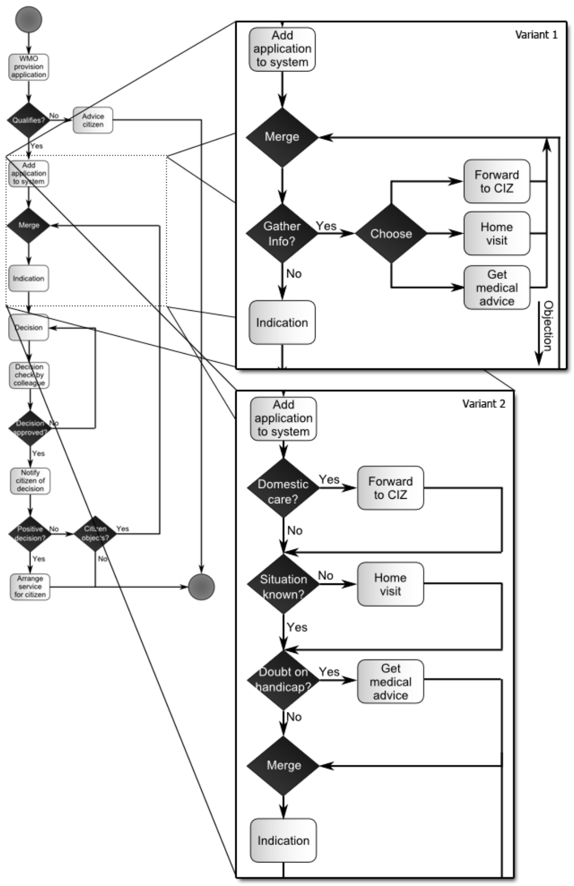


Figure 2: Imperative view on variability within the WMO processes.

illustrated in Figure 2, which ranged from all three activities being included into a single home visit, them being in strict sequential order to being in a loop, and so on. Other variability discovered included background checks on citizens, checks in the objection loop, different application mechanisms for different requests, and a decision check by a supervisor.

Figure 2 illustrates a traditional imperative variability view concerning the information gathering activities of the WMO process. Here two variants are shown which may or may not be used within the general template. Variant 1 contains the information gathering activities in a loop, whereas Variant 2 contains them in a sequence. However, not all three information gathering activities are required to be included or, in the case of the serial variant, not all are required to be traversed in the same order, leading to a large number

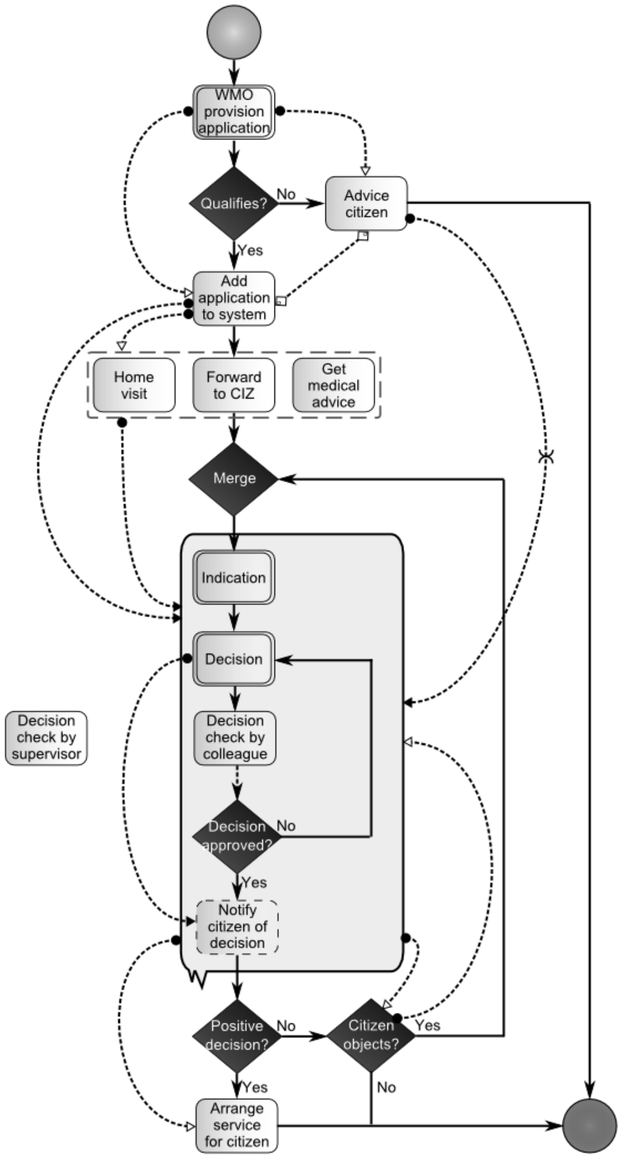


Figure 3: Declarative PVDI view on variability within the WMO processes.

of subvariants and increasing complexity from the design perspective. Figure 3, on the other hand, illustrates a declarative solution to the same WMO process using PVDI. Here, the declarative constraints are hidden by a set of graphical PVDI elements, which we explain in detail in Sections 4 and 5. Different arrows signify different ordering requirements, different borders signify different mandatory requirements, and the large group—called a frozen area—signifies an entirely locked area or one with limited variability. The flexibility of a process bounded with such constraints can vary from zero to unlimited, when there are no constraints at all. Notice how the constraints are not evaluated for templates but only for variant processes resulting from templates. Using a so called frozen area—which emulates an imperatively specified area within the process— we restrict the decision making process, which must be kept intact at all times. The rest of the template is captured using simple ordering constraints and one parallel constraint. Using this approach, variability is not added specifically to, for example, the information gathering activities, but instead they are not constrained and thus may or may not be added in any way possible. A special case of this can be seen in the large group where a dotted arrow signifies a place where the supervisor decision check activity may be inserted.

3 Process Variability through Declarative and Imperative techniques

Process Variability through Declarative and Imperative techniques (PVDI) aims at allowing a high degree of process variability while preserving the main business goal of a process. PVDI accomplishes this through combining blueprinting and constraint techniques [3]. The hard to understand constraints are being hidden from the user through easy to recognize graphical elements introduced to the blueprints or template. Because of this, PVDI allows the use of both declarative and imperative variability modeling techniques.

Figure 4 describes the engineering process of PVDI. On the left the process design part and on the right the template evolution part. The dark activities represent the well-known BPM engineering steps, whereas the light activities are PVDI specific ones. Template design is driven by requirements, business goals, laws, regulations, etc. The template modeler uses this knowl-

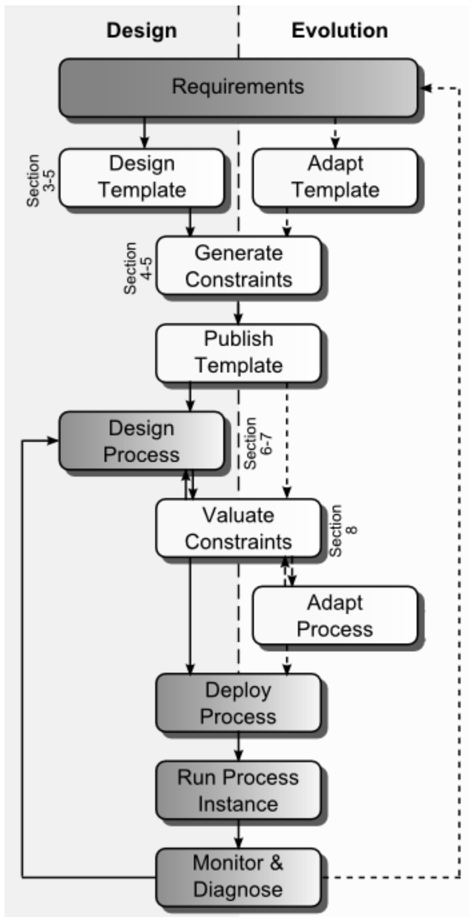


Figure 4: PVDI Engineering Process.

edge to model a template using traditional BPM techniques in combination with the graphical elements introduced through PVDI (Sections 3-5). Once completed, constraints are generated from the PVDI graphical elements and embedded within the template (Sections 4-5). The template is then offered to process modelers as a resource. A process modeler may use this template to model a variant of the process using traditional BPM techniques, with the aid of the graphical PVDI elements which visually point the modeler to what is, and what is not allowed (Sections 6-7). When satisfied, the resulting process is evaluated using the constraints within the template (Section 8). If the validation returns faults, the graphical PVDI elements related to the faults are highlighted for easy reference. Once the modeler completes a valid variant, the variant may be deployed, executed, and monitored as usual.

Whenever requirements drive towards template updates, the evolutionary cycle is entered. The template is updated, new constraints are generated, and the template is published again. At this point, whenever an existing variant based upon the previous version is run, a version check makes sure if the variant is up to date. If it is not, the variant is reevaluated with regards to the updated template. If this process returns faults, the variant is adapted to adhere to the new version of the template either automatically or manually. Once found correct, it may be redeployed, run, and monitored as before. We now introduce PVDI more precisely.

A process in PVDI is defined as a directed graph consisting of activities, gates, and events. Every activity contains multiple incoming and exactly one outgoing transition. The unique start and end event, contain exactly zero and multiple incoming transitions and one and zero outgoing transitions, respectively. Gates on the other hand may contain multiple incoming and outgoing transitions. Gates, as such, allow not only for simple branching, but also allow loops to occur within the process. Since a process is defined as a directed graph, it can serve as a framework for a modal logic of processes, including computational tree logic$^+$(CTL^+) [9]. Consider for example the process depicted in Figure 5. This process P consists of four activities A through D, a start event, an end event, and a gate.

Definition 3.1 (Process). *A process P is a tuple $\langle S, T \rangle$ where:*

- $S = (A \cup G \cup E)$ *is the set of activities, gates, and events;*
- $A = \{A_1 \dots A_n\}$ *is a finite set of activities;*
- $G = G_a \cup G_x$ *is a finite set of gateways, consisting of and– and xor–gates as defined by BPMN [10];*
- E *is a set of events, containing a unique start* $\odot$ *and end* $\otimes$ *event;*
- T *is a binary relation on* S*, i.e. a subset of the Cartesian product* $S \times S$*;*
- $\forall s \in S : (s, \odot), (\otimes, s) \notin T$*;*
- $\forall a \in A \cup E$ *there is at most one* $s \in S$ *with* $(a, s) \in T$*;*
- $\forall s \in S \setminus \{\otimes\}$ *there is at least one* $s' \in S$ *with* $(s, s') \in T$.

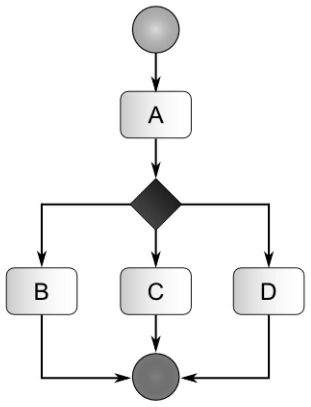


Figure 5: An abstract process.

Constraints are used by PVDI to capture restrictions on the variability offered within a PVDI template. These restrictions include, but are not limited to, process critical node and path information. A constraint in PVDI is a Computational Tree Logic$^+$ (CTL^+) formula [9, 11] (See Appendix A). Although any CTL^+ formula can be expressed in regular CTL, and it thus is equivalent to CTL, it does allow for compacter and better understandable formulas. In order to use a process as a model for the CTL^+ constraints, we introduce a set of variables and a valuation function. We use the **natural** valuation, that is, for each node $s \in S$ of a process we introduce a dedicated variable. The labeling function L is then built such that each dedicated variable is evaluated to *true* for its corresponding node only. Additionally, under the natural valuation we can use the same letter to represent both activity and its corresponding variable. When evaluated, every constraint must valuate to *true* at every node $s \in S$ of a process.

Definition 3.2 (Constraint)**.** *A constraint ϕ over a process P is a computation tree logic$^+$ (CTL^+) formula whose propositional variables are in $L(S)$ of P, where L is a labeling function using the natural valuation.*

From a notation point of view, we use Φ to denote the set of constraints related to process P. A constraint is valid for a process P iff it is evaluated to *true* in every node of the process under the natural valuation. More formally,

Definition 3.3 (Constraint validity)**.** *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 nodes* *of the process P. Then ϕ is* valid *iff* $\forall s \in S : \mathcal{M}, s \models \phi$.

Processes, defined in Definition 3.1, allow two different branching mechanisms as denoted by the two different gates. CTL^{+}, however, can not distinguish between these different branching mechanisms. As such, the constraints presented here enforce process structure instead of process execution.

Templates are used in PVDI as the basis for forming variants. Informally, a template is a process including constraints. Taking advantage of these formulations, we note that is possible to define underspecified processes. That is, a template may range from being a fully specified process, as defined in Definition 3.1, to a set of nodes S with $T = \emptyset$. As such, the inclusion of nodes and transitions in templates serve only the goal of constraint generation, and otherwise may be omitted if not to serve the goal of informally guiding variant design. Since a template is not a process, any constraints $\phi \in \Phi$ within a template do not have to valuate to *true* for that template but only for its variants.

Definition 3.4 (Template)**.** *A template R is a tuple $\langle S, T, \Phi \rangle$*

- *S as in definition 3.1;*
- *T is a binary relation on S, i.e. a subset of the Cartesian product $S \times S$;*
- $\forall s \in S : (s, \odot), (\otimes, s) \notin T$;
- $\forall a \in A \cup E$ *there is at most one* $s \in S$ *with* $(a, s) \in T$.
- Φ *is a finite set of constraints as defined in Definition 3.2.*

Variants in PVDI are processes which are based on a template and for which all constraints $\phi \in \Phi$ of that template are valid. A process based on a template for which not every constraint $\phi \in \Phi$ is evaluated to *true* at every node $s \in S$ of the process is therefore considered to not be a variant. When adding additional constraints to define templates, the set of possible variants is reduced. Note that, to support optimal variability, the sets of states and transitions of a template and its variant require no direct relation. This relation, instead, is implied through the constraints contained in the template and how they are required to valuate to true at every node of the variant. As a result, we allow any nodes and transition from templates to be removed in variants as long as all constraints validate to true.

Definition 3.5 (Variant)**.** *A variant $V = \langle S_V, T_V \rangle$ of a template $R = \langle S_R, T_R, \Phi_R \rangle$ is a process P such that Φ_R is valid for V.*

To ease the definition of constraints, we introduce the notion of a group in PVDI. That is a shorthand to describe a constraint spanning over a set of related nodes, in other words, to quantify a constraint over more than one node with different names.

Definition 3.6 (Group)**.** *A group G in the template $R = \langle S_R, T_R, \Phi \rangle$ is a nonempty subset of the set of nodes S_R of the template R. When a group s_g is used as input for a constraint instead of a single state s, then all occurrences of that single state s in the CTL^+ formula are replaced by $(s_1 \vee \ldots \vee s_n)$ where $s_1 \ldots s_n$ are elements of the group s_g.*

An example of a group can be seen in Figure 3 where it is used to group the home visit, forward to CIZ, and medical advice activities.

4 Template Design

In order to support template design, PVDI expands the graphical language of the Business Process Modeling Notation (BPMN) [10] with PVDI template elements. Each element consists of two parts: a *graphical element* which extends BPMN and a *constructive definition* which explains how to translate the graphical element into a constraint described in CTL^+. Template design in PVDI consists of several steps, illustrated in Figure 6. The dark steps are mandatory, and the light step is optional. Template design is naturally driven by requirements, such as those on the selection of activities, the order of activities, the importance of an entire sub-process, etcetera. Using this information, the template modeler selects a set of activities for the template. Then, any transition may be added to the template. These transitions are largely optional, and may be used to guide the modeling of variants. However, they may also be used for several advanced constraint techniques where entire sub-processes are being described using constraints. At the same time, PVDI elements are added to the template. These elements can be seen as being graphical representations of constraints and include, but are not limited to, mandatory selection and ordering between activities. Next, the actual CTL^+ constraints are generated from the PVDI elements introduced in the previous step and embedded in the template. Finally, the template is published for use as a source for variants.

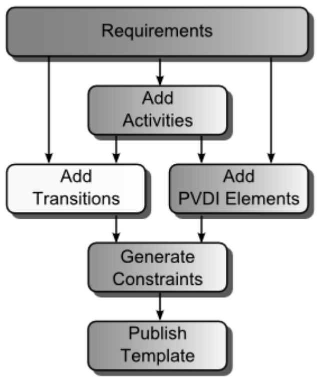


Figure 6: Template creation.

Let us consider these PVDI graphical elements and their translation into CTL^+ constraints one by one. The elements described here are not a final set and may by extended according to modeling needs. Since PVDI is in its design a declarative approach, first a number of elements using declarative techniques are discussed. Then, a number of elements more familiar to imperative techniques are introduced using PVDI's declarative approach.

4.1 Declarative Techniques

Declarative techniques are process flexibility techniques which focus on what tasks are performed [5]. One inherent property of declarative techniques is that every variation is allowed, except for what is specifically disallowed. Because of this, declarative techniques are considered the more flexible of the approaches, but less strictly defined. Here we define four declarative PVDI elements, their graphical representations, and their translation to CTL^+ constraints. The elements discussed here are Mandatory Selection, Mandatory Execution, Ordered Execution, and Parallel/Exclusive Execution.

4.1.1 Mandatory Between

The mandatory between constraint gives the option of marking nodes within a template in such a way that they must occur between two other nodes. Since mandatory between is used solely as a building block for other constraints, it does not have a corresponding graphical element. Mandatory between constrains the process using the CTL^+ formula $b \Rightarrow E([\neg e\, U\, s] \wedge Fe)$,

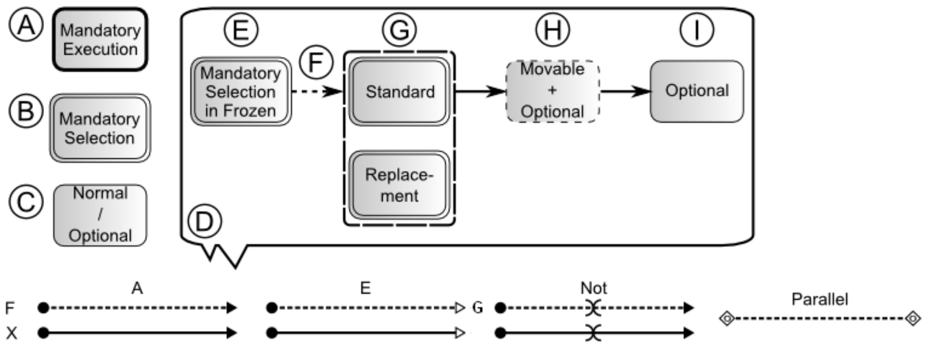


Figure 7: PVDI Graphical Elements.

meaning that at the begin node (b) there exists (E) a path for which we do not ($\neg$) encounter the end node (e) until (U) we encounter the marked node (s) and ($\wedge$) where we finally (F) encounter the end node (e). In other words, at b there is a path for which s comes before e. When a group is used instead of a single node, one of the nodes in the group must be encountered between b and e instead.

Definition 4.1 (Mandatory Between)**.** *The* mandatory between *is a constraint* $\alpha(b, e, s) = (b \Rightarrow E([\neg e\, U\, s] \wedge Fe))$.

4.1.2 Always Between

The always between constraint gives the option of marking nodes within a template in such a way that they must always occur in every path between two other nodes. Just like mandatory between, always between is used solely as a building block for other constraints, and therefore does not have a corresponding graphical element. Always between constrains the process using the CTL^+ formula $b \Rightarrow A([\neg e\, U\, s] \wedge Fe)$, meaning that at the begin node (b) for all (A) paths we do not ($\neg$) encounter the end node (e) until (U) we encounter the marked node (s) and ($\wedge$) we finally (F) encounter the end node (e). In other words, for all paths from b, s comes before e. When a group is used instead of a single node, one of the nodes in the group must always be encountered between b and e instead.

Definition 4.2 (Always Between)**.** *The* always between *is a constraint* $\beta(b, e, s) = (b \Rightarrow A([\neg e\, U\, s] \wedge Fe))$.

4.1.3 Not Between

The not between constraint gives the option of marking nodes within a template in such a way that they must never occur in any path between two other nodes. Just like the others, not between is used solely as a building block for other constraints, and therefore does not have a corresponding graphical element. Not between constrains the process using the CTL^+ formula $b \Rightarrow A[\neg s\ U\ e]$, meaning that at the begin node (b) for all (A) paths we do not ($\neg$) encounter the marked node (s) until (U) we encounter the end node (e). In other words, for all paths from b, we do not encounter s before e. When a group is used instead of a single node, none of the nodes in the group may be encountered between b and e instead.

Definition 4.3 (Not Between)**.** *The* not between *is a constraint* $\gamma(b, e, s) = (b \Rightarrow A([\neg s\ U\ e] \vee G\neg e))$.

4.1.4 Mandatory Selection

The mandatory selection constraint gives the option of marking nodes within a template in such a way that they must be selected for use in a variant. Any node which is not marked as mandatory is therefore considered to be optional.

Definition 4.4 (Mandatory Selection)**.** *A* mandatory selection *is a constraint* $\phi(s)$ *such that* $\phi(s) = \alpha(\odot, \otimes, s)$*, as described in Definition 4.1.*

A mandatory selection consists of the mandatory between where the start $\odot$ and end $\otimes$ events are used as the begin and end nodes of the mandatory between. As a result, the marked node must at least occur in one path between start and end events. In case a group is used instead of a single node, a disjunction is formed between the nodes within the group as defined in Definition 3.6, resulting in at least one of these nodes to occur within a path between the start and end events. The graphical representation of the mandatory selection element can be seen in Figure 7 as B, E, and the activities within group G. An example can be seen in Figure 3 where the application activity is marked as mandatory, and the indication and decision activities are marked as mandatory within their group.

4.1.5 Mandatory Inclusion

The mandatory inclusion element allows us to mark a node within a template as mandatory for execution path in every run–time instance of every variant.

In other words, every path must contain the marked node such that any execution path taken in a run–time instance of the variant encounters the marked node.

Definition 4.5 (Mandatory Inclusion)**.** *The* mandatory inclusion *is a constraint $\phi(s)$ such that $\phi(s) = \beta(\odot, \otimes, s)$, as described in Definition 4.2.*

A mandatory inclusion consists of the always between where the start $\odot$ and end $\otimes$ events are used as the begin and end nodes of the always between. As a result, all paths between the start and end events must include the marked node. When a group is used instead of a single node, at least one of the nodes in the group must occur in each path. Mandatory inclusion can thus be seen as a stricter version of mandatory selection (Definition 4.4). The graphical representation of the mandatory execution element can be seen in Figure 7 as A.

4.1.6 Ordered Execution

The ordered execution element allows to define the order of nodes in the template when used in the paths of the variants. An ordered execution is a relation between two nodes $p \in S$ and $q \in S$ stating the relative order of the two nodes in one or all execution paths. Groups of nodes may be used for both p and q, in which case *every* element within p must, or must not, be followed by *any* element within q. Note that neither p nor q becomes mandatory through the ordered execution. However, when p is included q could become mandatory as a result. The graphical representation of the ordered execution elements can be seen as flows at the bottom of Figure 7. Here, ordered execution is described over two dimensions; path and distance. The rows 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. Note that the negation of an ordered execution representing F results in the use of a G (Globally) instead for desired result. In Table 1, the corresponding CTL^+ formulas are displayed for each of the mentioned constraints. Here, the rows correspond to the CTL path quantifiers, and the columns to the CTL state quantifiers plus the optional negation. The formulas in the first row define a relation between p and q where q should

Finally follow p in all paths, a path, no path, and not a path respectively. The second row defines the same relations but only for the neXt node instead. Examples of these can be seen in Figure 3 where a number of Finally Exists and Finally All flows were used.

Definition 4.6 (Ordered Execution)**.** *An* ordered execution *is a constraint* $\phi(p, q, \Omega, \Pi) = (p \Rightarrow \Omega\Pi q)$*, with:*

- p, q *are nodes* $s \in S$ *or non-overlapping groups;*
- $\Omega \in \{A, E\}$ *is a state quantifier;*
- $\Pi \in \{X, \neg X, F, \neg F\}$ *is a path quantifier.*

$\Omega\backslash\Pi$	X	$\neg X$	F	$\neg F$
A	$p \Rightarrow AXq$	$p \Rightarrow AX\neg q$	$p \Rightarrow AFq$	$p \Rightarrow AG\neg q$
E	$p \Rightarrow EXq$	$p \Rightarrow EX\neg q$	$p \Rightarrow EFq$	$p \Rightarrow EG\neg q$

Table 1: Possible *Ordered Executions*

4.1.7 Parallel/Exclusive Execution

The parallel/exclusive execution element allows to enforce the non occurrence in the same path of two nodes. Parallel/exclusive execution constrains the process in such a way that from the two nodes $p \in S$ and $q \in S$ all paths (A) globally (G) may not encounter the other node. The result is that only one of the nodes p and q may be selected for use within a variant, or that they both must be preceded by a xor– or and–gate.

Definition 4.7 (Parallel/Exclusive execution)**.** *A* parallel/exclusive execution *is a constraint* $\phi(p, q) = (p \Rightarrow AG\neg q) \wedge (q \Rightarrow AG\neg p)$*, where* p, q *are distinct nodes* $s \in S$ *or non-overlapping groups.*

Note that any required precedence of a specific gate should be constrained by other means (ordered execution). When groups are used for p and/or q instead of a single node, every node within p must not be followed by any node within q and vice versa. The graphical representation of the parallel/exclusive execution element can be seen as a flow at the bottom right of Figure 7. An example of the parallel/exclusive execution can be seen in Figure 3 where it is used to require two different paths between the advice activity and the add application activity.

4.2 Imperative Techniques

Imperative techniques differ from declarative techniques by allowing no variations except for those which are specified beforehand. Because of this, imperative techniques are considered less flexible, but allow for an easy to use straightforward design process for variants. Given the nature of these techniques, we provide first two "areas" representing imperative specifications and then present several *modifications*, which change the flexibility of these areas in order to allow for variation points, that is, elements of a business process where change may occur to support imperative variability [3].

4.2.1 Closed Area

A closed area constrains the selected area in such a way that every node becomes mandatory to select (Definition 4.4) and no nodes other than those already in the area may be introduced to it. Closed areas allow exactly one incoming and exactly one outgoing flow to and from it.

Definition 4.8 (Closed Area)**.** *A* closed area *C over a group G is a set of constraints built over the set of activities $G_A = \{a_1, a_2, \ldots a_n\}$ of the group G and two dedicated start $\odot_C$ and end $\otimes_C$ nodes of the group. The set of constraints consists of the following:*

- Mandatory between *constraints $\alpha(\odot_C, \otimes_C, a_i)$ (Definition 4.1) for each activity $a_i \in G_A$;*
- *For the group $G_A^{-1} = S \setminus G_A$:*
    - $G_A^{-1} \Rightarrow \neg EX(a_1 \vee a_2 \vee \ldots \vee a_n \vee \otimes_C)$;
    - $(a_1 \vee a_2 \vee \ldots \vee a_n \vee \odot_C) \Rightarrow \neg EXG_A^{-1}$;
- *A* closed *constraint described by the CTL^+ formula $\odot_C \Rightarrow A[(\odot \vee a_1 \vee a_2 \vee \ldots \vee a_n)\ W\ \otimes_C]$, where $a_1, a_2, \ldots a_n$ are members of the set G_A.*

4.2.2 Frozen Area

Frozen areas specify areas in a template which may not be altered when designing variants unless specifically allowed. A frozen area constrains a part of a template in such a way that every node becomes mandatory to select (Definition 4.4) and that every path from every node allows for no variation until the exit of the area. Effectively, every path between its start and end is "frozen" and may not be changed. A frozen area is defined over a group (Definition 3.6) which may be used as such with regard to any other techniques described in this paper. The graphical representation of the frozen area can be seen in Figure 7 as D. An example of a frozen area can be seen in Figure 3 where it is used to limit variability in the decision making process.

Definition 4.9 (Frozen area)**.** *A* frozen area *F over a group G is a set of constraints built over the set of activities $G_A = \{a_1, a_2, \ldots a_n\}$ of the group G and two dedicated start $\odot_F$ and end $\otimes_F$ events of the group. The set of constraints consists of the following:*

- Mandatory between *constraints $\alpha(\odot_F, \otimes_F, a_i)$ (Definition 4.1) for each activity $a_i \in G_A$;*
- *For the group $G_A^{-1} = S \setminus G_A$:*
  - $G_A^{-1} \Rightarrow \neg EX(a_1 \vee a_2 \vee \ldots \vee a_n \vee \otimes_F)$;
  - $(a_1 \vee a_2 \vee \ldots \vee a_n \vee \odot_F) \Rightarrow \neg EXG_A^{-1}$;
- Path *constraints described by the CTL^+ formula $a_i \Rightarrow A(\pi_i^1 \vee \ldots \vee \pi_i^{n(i)})$ for each activity $a_i \in G_A \cup \{\odot_F\}$, where:*
  - *each of the sub–formulas π_i^j corresponds to a single path p_i^j leading from the activity a_i to the end of the group $\otimes_F$. There are as many sub–formulas π_i^j as there are distinct paths leading from a_i to the end;*
  - *for each of such paths $p_i^j = \{a_i, p_1^j, \ldots p_k^j, \otimes_F\}$, the corresponding sub–formula π_i^j is described by: $\pi_i^j = [(a_i \vee p_1^j \vee \ldots \vee p_k^j)\ W \otimes_F] \wedge [F\ p_1^j] \wedge \ldots \wedge [F\ p_k^j]$;*
  - *for the case of a simple path $p_i^j = \{a_i, \otimes_F\}$, the sub–formula π_i^j is reduced to $\pi_i^j = X \otimes_F$.*

Example 4.1 (Frozen Area).

$$\begin{tikzcd}
 & & & c \arrow[r] & d \arrow[dr] & \\
\odot_F \arrow[r] & a \arrow[r] & b \arrow[ur] \arrow[dr] & & & \otimes_F \\
 & & & e \arrow[r] & f \arrow[ur] &
\end{tikzcd}$$

$$a \Rightarrow A([(a \vee b \vee c \vee d)\ W\otimes] \wedge\ [Fb] \wedge [Fc] \wedge [Fd]) \vee ([(a \vee b \vee e \vee f)\ W\otimes] \wedge [Fb] \wedge [Fe] \wedge [Ff])$$

Assume the process illustrated above is encoded as a set of CTL^+ formulas. The formulas for $\odot$, a, b are the longest because there exist two possible paths from them to $\otimes$. Therefore, according to Definition 4.9, the formula splits into two pieces, one for each path. Consider the upper branch of the process, and the path $p_a^1 = \{a, b, c, d, \otimes_F\}$. The formula π_a^1 for that path is therefore the following: $[(a \vee b \vee c \vee d)\ W\ \otimes_F] \wedge [Fb] \wedge [Fc] \wedge [Fd]$. The same applies to the bottom branch. The resulting path–preserving formula for the activity a is illustrated in the figure above.

4.2.3 Optional Nodes

Allowing nodes to be optional is a modification of the definition of a frozen (Definition 4.9) area which allows for the removal of otherwise mandatory (Definition 4.4) nodes constrained by an area. Allowing for optional nodes modifies the constraints of the area in such a way that the affected nodes are no longer mandatory to select or may be bypassed within the area and thus become optional. The graphical representation of the optional nodes within a area can be seen in Figure 7 at H and I. An example of optional nodes can be seen in Figure 3 where both the decision check by a colleague and the gate are marked as being optional.

Modification 1 (*Allow for Optional Node*). Allowing for an optional node s is accomplished through the following modifications on the constraints:

- The *mandatory selection* constraint $\phi(\odot_F, \otimes_F, s)$ corresponding to the activity s is removed;
- All of the path sub–formulas π_i^j are modified in the following way: if the clause $[F\ s]$ is a part of the formula π_i^j, then that clause is removed

from the formula π_i^j.

4.2.4 Inserting Nodes

Allow node insertion is a modification of the constraints of a frozen area (Definition 4.9) which allows for the insertion of nodes at specific spots within a path constrained by a frozen area. The graphical representation of the option to insert nodes at spots within a frozen area can be seen in Figure 7 as the arrow at F. An example of the possibility to insert nodes can be seen in Figure 3 where a point within the path is left for the supervisor decision check activity.

Modification 2 (*Allow Node Insertion*). Allowing for the insertion of a node in a certain spot which lies between the activities a_t and a_{t+1} is accomplished with the following modifications:

- for all paths $p_i^j = \{a_i, a_{i+1} \dots a_t, a_{t+1}, \dots \otimes_F\}$ which contain both a_t and a_{t+1}, the corresponding path formula π_i^j is modified in the following way:
- $\pi_i^j = [(a_i \vee a_{i+1} \vee \dots \vee a_t)\ W\ AF\ \psi] \wedge [Fa_{i+1}] \dots \wedge [Fa_t]$, where ψ is the path–preserving formula for the activity a_{t+1} according to the definition of frozen area (Definition 4.9);
- note that the formula ψ may be in turn modified because of the existence of more places which allow for node insertion.

4.2.5 Moving Nodes

Allow node movement is a modification of the constraints of a frozen area (Definition 4.9) which allows a node to be moved within that area. Note that allowing a node to be moved does not automatically mean that it has a place to be moved to. Nor does it make the node optional. The graphical representation of the option to move nodes within a frozen area can be seen in Figure 7 at H. An example of a movable node can be seen in Figure 3 where the notification activity is marked as being movable within the group it is contained. An extra ordered execution flow (Definition 4.6) is however added to ensure it always following the decision activity.

Modification 3 (*Allow Node Movement*). Allowing for the moving of a node s is accomplished with the following modifications: the path–preserving formula for the node s of the type $s \Rightarrow \phi$ is removed.

4.2.6 Replacing Nodes

Allow node replacement is a modification of the constraints of a frozen area (Definition 4.9) which allows the choice to include one of several nodes at a certain point in a path of a frozen area. The graphical representation of the option to replace nodes within a frozen area can be seen in Figure 7 at G.

Modification 4 (*Allow Node Replacement*). Allowing for the replacement of node s_j^0 with nodes $s_j^1, \ldots, s_j^i$ in a path is accomplished through the following modification of mandatory and path–preserving constraints:

- Replace every occurrence of s_j^0 in every constraint with the following clause: $(s_j^0 \vee s_j^1 \vee \ldots \vee s_j^i)$.

4.2.7 Swapping Nodes

Allow node swapping is a modification of the constraints of a frozen area (Definition 4.9) which allows the choice to swap certain nodes. Allowing for the swapping of nodes s_j and s_i modifies the frozen area in the same way as allowing for their replacement (Modification 4.2.6). However, in the case of swapping nodes the mandatory selection (Definition 4.4) is not altered. The graphical representation of the option to swap nodes within a frozen group is the same as the replacement of nodes and can be seen in Figure 7 at G.

Modification 5 (*Allow Node Swapping*). Allowing for the swapping of nodes s_j and s_i is accomplished through the modification of Π in Definition 4.9 with the following:

- Allow for node replacement with s_j and s_i at nodes s_j and s_i as described by Modification 4.2.6, but modify the path constraints only.

5 Constraint Relations

So far we have introduced constraints as CTL^+ implications. Although these offer a considerable amount of expressivity for variability, we need more complex constructions to capture other important variability functionality. COVAMOF for example allows relations between different variation points at different levels of abstraction. These so called realization relations *"specify rules that determine which variants or values at variation points at lower levels should be selected in order to realize the choice at variation points at higher levels"* [2]. Although PVDI does not feature variation points, a similar mechanism is supported in the form of Constraint Relations. A constraint relation is a constraint which forms a zeroth–order logic formula over two other constraints, that is, a formula without quantifiers. In [19] a number of selection constraints were defined that are ideal to include in PVDI through constraint relations. Next, we redefine these selection constraints taken from [19], the prerequisite, exclusion, substitutions, corequisite, and exclusive-choice as constraint relations.

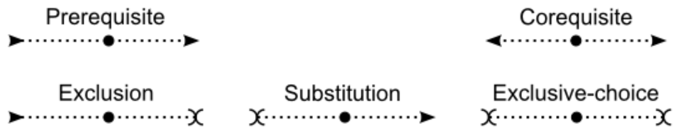


Figure 8: PVDI Graphical Elements for Variation Relations.

5.1 Prerequisite

The prerequisite constraint relation defines an affiliation between two nodes regarding their inclusion, but without any requirements on ordering. The prerequisite relation constrains the process in such a way that if p is included q must be included as well. When a group is used for p or q, then when at least one node from p is included, at least one node from q must be included as well. The graphical representation of the prerequisite relation can be seen in Figure 8 at the top left.

Definition 5.1 (Prerequisite)**.** *A* prerequisite *constraint* $\phi(p, q) = (\alpha(\odot, \otimes, p) \Rightarrow \alpha(\odot, \otimes, q))$*, as described in Definition 4.1, with* p, q *being different nodes* $s \in S$ *or non overlapping groups.*

5.2 Exclusion

The exclusion constraint relation defines an affiliation between two nodes regarding the exclusion of one of them. The exclusion relation constrains the process in such a way that if p is included q must not be included. When a group is used for p or q, then when at least one node from p is included, no node from q may be included. The graphical representation of the exclusion relation can be seen in Figure 8 at the bottom left.

Definition 5.2 (Exclusion)**.** *An* exclusion *constraint* $\phi(p, q) = (\alpha(\odot, \otimes, p) \Rightarrow \gamma(\odot, \otimes, q))$*, as described in Definition 4.1 and Definition 4.3, with* p, q *being different nodes* $s \in S$ *or non overlapping groups.*

5.3 Substitution

The substitution constraint relation defines an affiliation between two nodes regarding their substitution. The substitution relation constrains the process in such a way that if p is not included q must be included instead. When a group is used for p or q, then when no node from p is included, at least one node from q must be included. The graphical representation of the substitution relation can be seen in Figure 8 at the bottom in the middle.

Definition 5.3 (Substitution)**.** *A* substitution *constraint* $\phi(p, q) = (\gamma(\odot, \otimes, p) \Rightarrow \alpha(\odot, \otimes, q))$*, as described in Definition 4.1 and Definition 4.3, with* p, q *being different nodes* $s \in S$ *or non overlapping groups.*

5.4 Corequisite

The corequisite constraint relation defines an affiliation between two nodes regarding their inclusion. The corequisite relation constrains the process in such a way that if p is included q must be included as well, and vice versa. When a group is used for p or q, then when at least one node from p is included, at least one node from q must be included as well, and vice versa. The graphical representation of the corequisite relation can be seen in Figure 8 at the top right.

Definition 5.4 (Corequisite)**.** *A* corequisite *constraint* $\phi(p, q) = \psi(p, q) \wedge \psi(q, p)$*, where* ψ *is the* prerequisite *constraint (Definition 5.1) and* p, q *are different nodes* $s \in S$ *or non overlapping groups.*

5.5 Exclusive–Choice

The exclusive–choice constraint relation defines an affiliation between two nodes regarding their inclusion and exclusion. The exclusive–choice relation constrains the process in such a way that if p is included q must not be included, and vice versa. When a group is used for p or q, then when at least one node from p is included, no node from q may be included, and vice versa. The graphical representation of the exclusive–choice relation can be seen in Figure 8 at the bottom right.

Definition 5.5 (Exclusive–Choice)**.** *An* exclusive–choice *constraint* $\phi(p, q) = \psi(p, q) \wedge \psi(q, p)$*, where* ψ *is the* exclusion *constraint (Definition 5.2) and* p, q *are different nodes* $s \in S$ *or non overlapping groups.*

6 Process Healthiness

There are a number of possible metrics to verify if a business process model is healthy or not. In [12] three main characteristics of a healthy, or sound, processes are mentioned. Namely, the (weak) option to complete, proper completion, and the absence of "dead" transitions. In PVDI, those characteristics are ensured through the help of the healthiness constraint.

Definition 6.1 (Healthiness)**.** *The* healthiness *is a constraint* $\phi(s) = \alpha(\odot, \otimes, s)$*, as described in Definition 4.1.*

The healthiness constraint disallows dead–ends and ensures that all nodes are reachable at the same time (Definition 6.1). It consists of a mandatory between where the start $\odot$ and end $\otimes$ events are used as the begin and end nodes of the mandatory between. In contrast with the constraints discussed in previous sections, which are generated from the template before modeling a variant, the constraints described in this section are generated after modeling a variant and directly prior to validation. Although the constraint seems equal to the mandatory selection (Definition 4.4), the difference in the time of generation allows for different uses. A process (Definition 3.1) is considered healthy when all healthiness constraints have been evaluated to *true* at all nodes of the process.

Definition 6.2 (Healthy Process)**.** *A process* P *is healthy iff* $\forall s \in S_p$ *the healthiness constraint* $\phi(s)$ *is valid at every node* $s \in S_P$ *of the process.*

7 Variant Validation

Processes are required to be validated after derivation from a template. A process is *valid* with respect to a template if it is healthy (Definition 6.2) and if it is a variant (Definition 3.5) of this template.

Definition 7.1 (Valid Process)**.** *A process P is* valid *with respect to a template R iff it is a variant of R and is healthy.*

As specified in Definition 3.5, a process P is a variant of a template R if the set of constraints Φ_R is valid for P. The same is true for healthiness (Definition 6.2), P is healthy if the set of healthiness constraints Φ_H is valid for P. In turn, according to Definition 3.3, these constraints are valid if $\forall s \in S_P : \mathcal{M}, s \models \Phi_R \cup \Phi_H$. As such, validation entails that the process is evaluated against these sets of constraints. We are therefore in need of an algorithm which valuates every constraint for the model $\mathcal{M}$.

7.1 Model Conversion

Model checking is a technique used to automatically verify models against a given specification. In classical model checking (e.g., [13]), a model is defined as a finite state machine, and is checked against a set of formulas of propositional or modal logic. In case of PVDI, a process, which is defined as a directed graph (Definition 3.1), is validated against a set of constraints expressed as CTL^+ logic formulas. We therefore employ model checking techniques when verifying variants [14]. A variant is defined as a process $P = \langle S, T \rangle$ for which all constraints are valid. A CTL model $\mathcal{M} = \langle S, T, L \rangle$ consists of a set of states S, a set of transitions T, and a valuation function L [9,11,14]. In order to get the corresponding model $\mathcal{M}$ of P, we map S and T such that loops are mapped once, but infinite traversals of loops are avoided. Since we evaluate business processes which at most might be long living, but never infinite, any path, being a sequence $(s_0, s_1, \ldots)$ of states such that $(s_i, s_{i+1}) \in T$, can therefore only be *finite*. And lastly, to define the labeling function L we use natural valuation, that is, for each node $s \in S$ of the process we define a dedicated variable which evaluates true at that node only. For ease of reference, we name this variable the exact same as the node itself. Individual CTL^+ constraints are evaluated on the model $\mathcal{M}$ using state space enumeration. Then, a constraint ϕ is valid for the process

P iff $\forall s \in S : \mathcal{M}, s \models \phi$, where $\mathcal{M}$ is the corresponding model of P (Definition 3.3). Now we specified a model, an algorithm supporting CTL^+ model checking for directed graphs using natural valuation is required.

7.2 Validation Algorithm

Although many model checkers exist, we specified a simple search algorithm using state space enumeration to test the feasibility of model checking business processes. In doing so, we did tailor the algorithm for the specific use of business processes with finite paths. But, although the results are positive, we are certain that great advances in computation time can be achieved through the introduction of a more efficient algorithm. The validation algorithm is also a part of our full–featured demo, which supports the visual modeling of business process templates and the validation of the variants [15].

The validation algorithm is implemented through a package containing classes with a one-to-one mapping of the CTL^+ symbols described in Appendix A. As a result, any correct CTL^+ formula is supported by the algorithm, enabling easy extensibility of the set of constraints described earlier. The core algorithm consists of

- StateQuantifiers;
  - All, Exists;
  - Implies, Proposition;
  - Or, And, Negation.
- PathQuantifiers;
  - Next, Finally, Globally, Until, Unless;
  - Or, And, Negation.

The StateQuantifiers All and Exists take a single PathQuantifier as argument. Implies takes two StateQuantifiers as arguments, and Proposition —which here resembles an atomic formula— takes a node or node type as argument. The PathQuantifiers Next, Finally, and Globally take a single StateQuantifier as argument, whereas Until and Unless take two. The quantifiers Or, And, and Negation take their own type as input only. Through these specific interactions tree-like constructions representing only correct CTL^+ formulas can be formed. As an example, the CTL^+ formula $p \Rightarrow A[qUr]$ would be constructed like so $Implies(Proposition(p), All(Until(Proposition(q), Proposition(r))))$.

Both StateQuantifiers and PathQuantifiers implement validate methods which move through a process tree employing the validate methods of its children until a correctness decision is reached. We discuss the validate methods of the non-trivial core elements. To increase readability, these methods lack those lines and arguments used for optimization and a number of safeguards, but remain the same otherwise.

Listing 1: Validate Method of the StateQuantifier All

```
public boolean validate(CTLNode e){
  Iterator<List<CTLNode>> paths =
    CTLUtil.getAllPathsFromNode(e).iterator();
  boolean ret = paths.hasNext();

  while(paths.hasNext() && ret)
    ret = q.validate(paths.next());

  return ret;
}
```

Listing 1 illustrates the validate method for the All StateQuantifier. The validate method takes a node e of the process tree as input. It then requests all paths from this node e and initializes its variables. In such a case that there are no paths returned the method returns false. However, in practice the paths returned will always include the CTLNode e as its first element and therefore should never be empty. For all paths, the validate method of the PathQuantifier child element q is called until one returns false. When all paths return positively, return true, and false otherwise.

Listing 2: Validate Method of the StateQuantifier Exists

```
public boolean validate(CTLNode e){
  Iterator<List<CTLNode>> paths =
    CTLUtil.getAllPathsFromNode(e).iterator();
  boolean ret = false;

  while(paths.hasNext() && !ret)
    ret = q.validate(paths.next());

  return ret;
}
```

Listing 2 illustrates the validation method for the Exists StateQuantifier. The validate method operates in the same manner as that of the All StateQuantifier, but returns true as soon as one path returns true.

Listing 3: Validate Method of the PathQuantifier Next

```
public boolean validate(List<CTLNode> path){
  return path.size() > 1 && p.validate(path.get(1));
}
```

Listing 3 illustrates the validate method for the Next PathQuantifier. The validate method receives a path as input, checks if a next element exists, calls the validate method of its StateQuantifier child element *p* for that next element, and returns the result.

Listing 4: Validate Method of the PathQuantifier Finally

```
public boolean validate(List<CTLNode> path){
  boolean ret = false;
  Iterator<CTLNode> pathIt = path.iterator();

  while(pathIt.hasNext() && !ret)
    ret = p.validate(pathIt.next());

  return ret;
}
```

Listing 5: Validate Method of the PathQuantifier Globally

```
public boolean validate(List<CTLNode> path){
  Iterator<CTLNode> pathIt = path.iterator();
  CTLNode n = null;
  boolean ret = pathIt.hasNext();

  while(pathIt.hasNext() && ret){
    n = pathIt.next();
    if(!(n instanceof CTLLoopNode))
      ret = p.validate(n);
  }
  return ret;
}
```

Listing 4 illustrates the validate method for the Finally PathQuantifier. The validate method receives a path as input, and initializes its variables. It then loops through the path and calls the validate method of its StateQuantifier child element p for each element until a positive result is returned. In this case the loop is interrupted and true is returned immediately. In case the end of the path is reached without a positive result false is returned. The validate method for the Globally PathQuantifier is depicted in Listing 5. It operates very much in the same way as the Finally PathQuantifier, except that it requires a positive result allong the entire path for a return of true. In cases where a loop is detected at the end of a path, p holds globally for the infinite loop and true is returned.

Listing 6 illustrates the validation method for the Until PathQuantifier. Only the Until PathQuantifier is discussed here as the Unless PathQuantifier is very similar. The validate method of the Until requires the StateQuantifier child element p to hold in the path until StateQuantifier child element q holds. After initializing its variables, the validate method loops through the path and calls the validate methods of both p and q. While the validate method of p returns positively, it continues looping through the path. When the validate methods of both p and q do not return positively, the loop is interrupted and false is returned. In case the validate method of q returns positively, and the validate method of p has returned positively so far, the loop is interrupted and true is returned. In all other cases false is returned.

Listing 6: Validate Method of the PathQuantifier Until

```java
public boolean validate(List<CTLNode> path){
  Iterator<CTLNode> pathIt = path.iterator();
  CTLNode n = null;
  boolean ret = false;
  boolean ok = pathIt.hasNext();

  while(pathIt.hasNext() && !ret && ok){
    n = pathIt.next();
    if(!(n instanceof CTLLoopNode)){
      ok = p.validate(n);
      ret = q.validate(n);
    }
  }
  return ret;
}
```

Finally, the proposition StateQuantifier is an abstract entity and is therefore not listed here. Instead several child elements of this quantifier exist. The most common of these is one which validates if an element in the process tree is located at the current node in the path. Others, for example, validate if the current node in the path is a start $\odot$ or an end $\otimes$ event. Using combinations of the elements discussed here, any correct $CTL+$ formula can be formulated and validated. And, as a result, the set of constraints and formulas discussed earlier can be easily extended upon due to the modular design of the validation algorithm.

7.3 Performance

A performance test of the validation algorithm was conducted using a machine with an Intel Core i7 950 at 3.07GHz, 6GB RAM (3×2GB Triple Channel), and an Intel SSD SA2M080G2GC running Windows 7 SP1 (64-bit) and Java 6 update 23 (64-bit). The performance test consisted of the evaluation of three different business processes consisting of 13 to 20 nodes, 15 to 21 transitions, and up to 1 frozen area. Each evaluation consists of the valuation of the in the business process embedded constraints, including soundness constraints, at every node. The number of constraints ranged from 18 to 28, where each frozen area counts as one constraint. Since frozen areas consist of a large number of constraints, the number of constraints for the two processes containing a frozen group was actually much higher. Table 2 contains the results of the constraint valuation performance test conducted with the VxBPMN tool. Every valuation was run one thousand times in order to get a fair average reading and took between 599 and 3525 milliseconds for all thousand runs. On average each individual process valuation took 3.5 milliseconds for test 1, 1.5 milliseconds for test 2, and 0.6 milliseconds for test 3. Although on first sight test 2 and 3 seem quite similar–except for the two constraint violations–, the difference of 1 millisecond can actually be explained by the difference in complexity of the process, which included a number of loops, and therefore included a much higher amount of paths. When looking at the average time required to valuate a single constraint, we notice that the slowest times occur at test 1 with $3525ms/1000/18 = 0.2$ milliseconds per constraint. The test with the lowest amount of nodes, transitions, frozen areas, and constraints therefore resulted in on the average the slowest valuation times per individual constraint. Test 1, however, included a high amount of paths due to a set of branches within a loop. From this we speculate that valuation performance scales well with regard to the amount of nodes and transitions included in variants, but less with complicated looping constructs.

	Time (ms)		**Model**			**Constraints**	
#	**Total**	**Avg**	**Nodes**	**Flows**	**Frozen**	**Total**	**Faults**
1	3525	3.5	13	15	0	18	0
2	1511	1.5	18	21	1	28	0
3	599	0.6	20	20	1	26	2

Table 2: Constraint valuation performance

8 Variant Design: An Example

The design of variants in PVDI is naturally eased by the PVDI graphical elements. Every element introduced in the previous sections guides the design towards a set of possible variants. Take for example the PVDI template depicted in Figure 9. This template specifies a simple room reservation process including three different rooms: practical labs, classrooms, and meeting rooms. Because the three rooms are grouped, and an ordered execution constraint specifies that the start element must be followed by the group, at least one type of room must be included, but more than one may be included. The group is then followed by a frozen area consisting of four activities: lock table, view rooms, reserve room, and error. Of these, two are mandatory to select, and two (lock and error) are allowed to be removed. However, by using an exclusive–choice constraint relation we specify that at least one of the two optional activities must be included. In this way, we force either a lock table before reserving mechanism, or a first come first serve mechanism including a success/error report in case of failure when the room was already reserved by another.

Activity	Variable	Activity	Variable
Start	s	Group end	e_g
End	e	Lock	lo
Practical Lab	pr	View Rooms	vr
Classroom	cr	Reserve Rooms	rr
Meeting room	mr	Check Error	ce
Group start	s_g		

Table 3: PVDI Room Reservation Elements.

Once a template is created, its PVDI elements are automatically con-

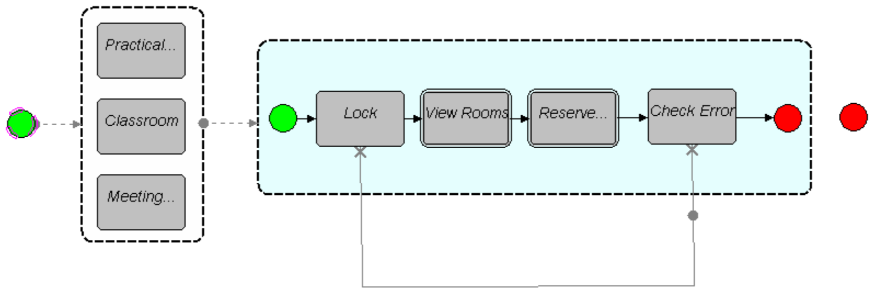


Figure 9: PVDI Room Reservation Template.

verted by the tool to a set of CTL^+ constraints as described in sections 4 and 5. The following constraints are generated from the PVDI template depicted in Figure 9. We use shorthand notations for each element, the meaning of which can be found in Table 3. Each line represents a single constraint. Lines 1 and 2 represent the flowconstraints leading to and from the group respectively. Lines 3 to 7 represent the pathing constraints for the frozen area. Lines 8 and 9 include the mandatory constraints for the two mandatory nodes in the area, and lines 10 and 11 list the restrictions on entering and exiting the area through only its start and end nodes. Finally, line 12 lists the exclusive–choice constraint. Note that the healthiness constraints are generated at a later point in the process, and are therefore not included here.

$$s \Rightarrow AF(pr \vee cr \vee mr) \tag{1}$$
$$(pr \vee cr \vee mr) \Rightarrow AF(s_g \vee lo \vee vr \vee rr \vee rr \vee ce \vee e_g) \tag{2}$$
$$s_g \Rightarrow A([(s_g \vee lo \vee vr \vee rr \vee ce)We_g] \wedge Fvr \wedge Frr) \tag{3}$$
$$lo \Rightarrow A([(lo \vee vr \vee rr \vee ce)We_g] \wedge Fvr \wedge Frr) \tag{4}$$
$$vr \Rightarrow A([(vr \vee rr \vee ce)We_g] \wedge Frr) \tag{5}$$
$$rr \Rightarrow A[(rr \vee cc)We_g] \tag{6}$$
$$ce \Rightarrow A[(ce)We_g] \tag{7}$$
$$s_g \Rightarrow E([\neg e_g Uvr] \wedge Fe_g) \tag{8}$$
$$s_g \Rightarrow E([\neg e_g Urr] \wedge Fe_g) \tag{9}$$
$$[s \vee e \vee pr \vee cr \vee mr] \Rightarrow \neg EX[lo \vee vr \vee rr \vee ce \vee e_g] \tag{10}$$
$$[s_g \vee lo \vee vr \vee rr \vee ce] \Rightarrow \neg EX[s \vee e \vee pr \vee cr \vee mr] \tag{11}$$
$$[\alpha(s,e,lo) \Rightarrow \gamma(s,e,ce)] \wedge [\alpha(s,e,ce) \Rightarrow \gamma(s,e,lo)] \tag{12}$$

Next, variants are designed using the template. An example of a valid business process variant is shown in Figure 10. There a practical lab is chosen and an exclusive–lock mechanism is used to avoid any collisions.

Figure 11 shows an erroneous example. Two PVDI elements are being highlighted by the tool, which indicates that their related formulas are being violated. First, none of three predefined room types is used, resulting in a violation of the constraint at line 1. Second, the frozen area structure is corrupted since the activity "Check Error" is placed before the activity "View Rooms", resulting in a violation of the constraint at line 7.

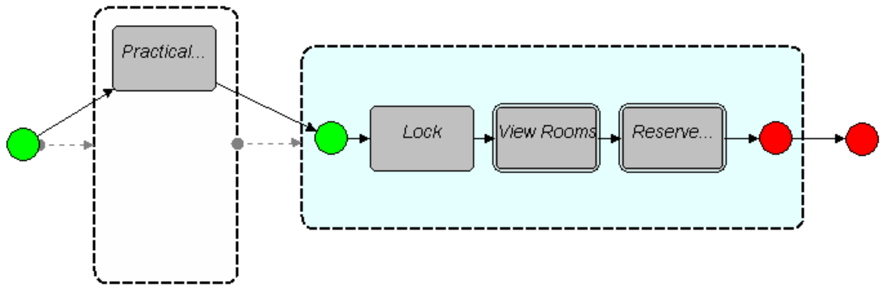


Figure 10: A Valid Variant of Room Reservation Process.

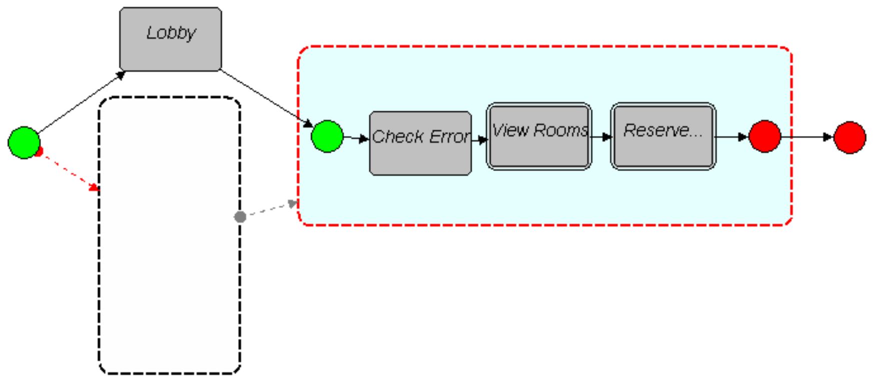


Figure 11: An Example of Invalid Variant for Room Reservation Process.

9 Evaluation

To evaluate the proposed PVDI, we compare it with the more common imperative techniques on the grounds of expressive power and complexity of design. Given that there are no metrics or benchmarks available, we therefore begin by providing a framework for evaluation. For defining the boundaries of the comparison, we give a definition of both the imperative variability and the declarative PVDI variability. We then define several template properties which we use to identify expressive power features. And finally, we explore the difference in complexity, intended as the number of variants that can be described compactly with one approach.

9.1 The Imperative Case

Imperative process specifications focus on a specific process definition by using transitions to prescribe the order of node traversal [16]. These structural variations adapt a process by applying a list of atomic operations in a specific order to the business process. Such operations for example include the replacement of an activity by another one, the addition of a flow, or the removal of a process fragment [3, 17]. Since different structural variations can contradict each other, it is necessary to specify which structural variations may or may not be applied together using *variation relations*. We call the combination of these two mechanisms *variation points*.

An example of a variation point is illustrated in Figure 14d. The upper branch contains two hexagonal tokens signifying a variation point where either activity "C" or activity "D" may be included.

Definition 9.1 (Imperative Template)**.** *An* imperative template *R is a tuple $\langle P, VP \rangle$ where P is a process and VP is a set of variation points.*

When one or more structural variations are selected from the template, the resulting process is called a *variant.* A variant may only contain structural variations as allowed by the variation relations of the different variation points in the template. A process containing combinations of structural variations not allowed by the variation relations is therefore not a variant. Imperative variability is the ability to produce a variant V by selecting a set of structural variations from a template.

9.2 The Declarative PVDI Case

Declarative process specifications define relations between tasks in the form of constraints, allowing any paths as long as these constraints are not violated [16]. PVDI is based on this approach, but with one important difference. Instead of interpreting constraints on the state space of the process graph, PVDI evaluates them on the graph itself. We use the PVDI definitions for constraints, templates, and variants as provided in Section 3 to evaluate the declarative expressive power of PVDI and to then compare it with the imperative case. Declarative variability as used by PVDI is the ability to produce multiple non-bisimilar variants from a template R for which every constraint $\phi \in R_\Phi$ evaluates to *true* at every node $s_i \in v_S$ of every variant v. Non-bisimilar variants consist of those variants which do not effectively simulate each others behavior and thus offer unique process flows [18].

9.3 Expressive Power

Templates are used in both variability techniques to capture a process plus the available variability. Given that we are interested in comparing the two approaches, we preliminary define the property of being finite and closed of a template.

Definition 9.2 (Finite and Closed Templates)**.** *A template R is* finite *iff it has only finitely many variants. It is* closed *iff the set of nodes of every variant of R is contained in the set of nodes of R.*

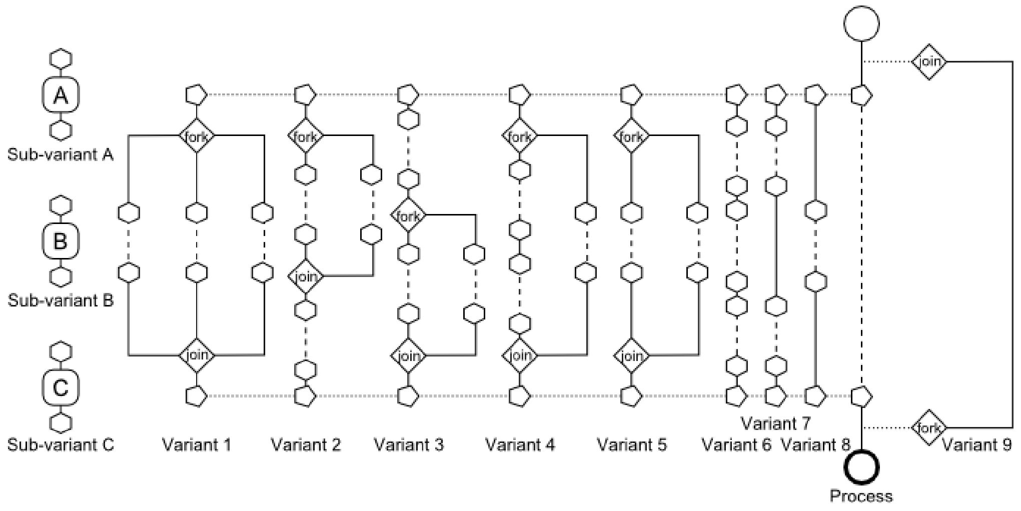


Figure 12: High variable imperative solution.

The second property describes closed templates. A template is closed if and only if for all possible variants based on that template, the set of nodes is a subset of the set of nodes in the template. In other words, no new node can be introduced to variants. This entails that any template which is closed is also finite, and thus offers only a limited and very specific set of variants as the following theorem.

Theorem 9.1. *Any template R that is closed is also finite.*

Proof. A closed template R produces a set of variants V such that $\forall v \in V : v_S \subseteq R_S$. Since $|R_S| \in N$, disregarding constraints, only a limited number of transitions V_T can be drawn between the nodes in R_S. Meaning $|V_T| \in N$. It follows that $|V| \in N$, and from the definition that R is finite. □

Imperative variability is expressed through templates which include variation points. The expressive power of imperative variability is therefore directly connected to the template itself and the available set of atomic operations [3, 17]. Theorem 9.2 shows that imperative templates are both closed and finite. As a result, all variability offered through imperative templates must be specifically designed and prescribed within the template.

Theorem 9.2. *All imperative templates are closed and finite.*

Proof. An imperative template R consist of a set of nodes R_S, transitions R_T, and a set of variation points R_{VP}. An imperative variant V consists of an imperative template V_R and a subset of structural variations $SV \subseteq VP_{SV}$ chosen from those in the variation points of the template $V_{R_{VP}}$. Therefore, all nodes in variants $V_S \subseteq R_S$. From the definition it follows that R is closed, and from Theorem 9.1 it follows that R is finite. □

Imperative variability frameworks however do sometimes include techniques which increase the expressive power of the framework, allowing for non-closed and non-finite templates. A common example is the so–called *placeholder node*. A placeholder is a place in a template which may or may not be used to include a new node. In PVDI terms it may be described as $p \Rightarrow (AXq \vee AXAXq)$, where the placeholder is preceded by p and followed by q. As a result, a template including such placeholders becomes not closed, nor finite, and thus more expressive. Other frameworks allow structural variations within structural variations, a powerful construct that can easily lead to inconsistent and unmanageable variants. Allowing this, does break the finite property from imperative templates, while the closed property does remain valid since all activities in the variant remain a subset of those included in the template.

Contrary to imperative variability, variability in PVDI is not defined explicitly within templates. Instead, the variability is offered through underspecification of the process. Those variability options which are not allowed are specifically disallowed through constraints. The expressive power of PVDI is therefore directly related to the ability to disallow one thing and allow others. In other words, to disallow exactly enough without allowing unwanted possibilities. Because of this method of underspecification we know from Theorem 9.3 that PVDI templates have the option of specifying templates in such a way that they are not closed, nor finite.

Theorem 9.3. *There exist PVDI templates that are not closed nor finite.*

Proof. Consider a PVDI template R, which consists of a set of nodes $R_S = \{p, q\}$, a set of transitions $R_T = \emptyset$, and a set of constraints $R_\Phi = \{p \Rightarrow AFq\}$. Variant V based on this template consists of the template $V_R = R$ a set of nodes $V_S = \{p, q, r\}$, and a set of transitions $V_T = \{p \to r, r \to q\}$. All constraints R_Φ valuate to *true* at all nodes $s_i \in V_S$. Because V is a variant and $V_S \nsubseteq R_S$ we conclude that R is not closed. Since we may replace $r \in V_S$ with any node and produce a variant, we also conclude R is not finite. □

Declarative frameworks sometimes include imperative techniques. Not to increase theoretical expressiveness, but practical expressiveness. For example, in [19, 20] authors use an imperative process structure with pockets of declarative variability, and in [6] we propose to capture imperative operations with sets of constraints. We extended our proposal in Section 4.2.

9.4 Ease of Use

It is difficult to make a clear comparison between imperative and declarative approaches regarding practical use in terms of their complexity. This is mainly due to the fact that their usability varies depending on each particular case and the particular variability tool or framework used. In practice, imperative approaches are useful when dealing with templates with limited flexibility. On the other hand, declarative templates offer a great deal of flexibility, which is useful in the case of highly variable business processes but turns out to give large overheads when dealing with templates with limited flexibility. Next, we define a framework for the comparison and we quantitatively compare the relative complexities of both approaches regarding their ease of use.

To given an impression of how this happens in practice consider the example in Figure 12 illustrating the issue arising when implementing a highly variable template utilizing imperative techniques. All the combinations shown in the figure must be implemented explicitly as nine variants, and each of those variants is made of variation points (shown as pairs of hexagons with dashed connector between them). In result, a process modeler should choose one of nine predefined variants, and then choose how to fill the placeholders with a variant of choice. The template includes more than 50 possible variations in total, which must be provided at the template level either explicitly or by allowing to fill the placeholders by different activities. The difficulty of the task even increases in situations where some of the combinations are not allowed. Such restrictions should be reflected by the template, for example, by linking a list of possible options to particular placeholders. In the worst case, each of the more than 50 possible options should be visited at the stage of template modeling in order to decide if it is allowed or not. On the other hand, the same task can be easily solved using PVDI techniques. Only two formulas are required in order to make the specification: $start \Rightarrow AF(A \vee B \vee C)$ and $(A \vee B \vee C) \Rightarrow AFend$. The PVDI equivalent of this can be seen in Figure 13. In order to disallow some combinations, additional constraints can be added in order to reflect the rules which restrict the possible customizations.

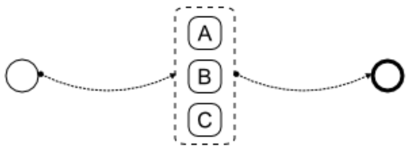
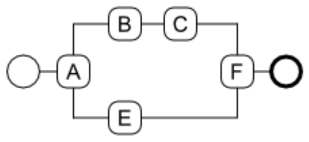


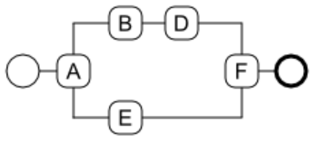
Figure 13: High variable declarative solution.

The situation changes when dealing with low–variable cases. Consider for instance the two possible variants shown in Figures 14a and 14b. They only differ in one activity: either C or D is included in the upper branch following the activity B. An imperative template is simple, as illustrated in Figure 14d, but it is more complicated while trying to convert this template into a set of formulas. The set of formulas in Figure 14c shows just one of possible options consisting of nine formulas. Solutions will obviously become rapidly more complicated for larger processes. These low– and high–variable cases offer valuable insights into the complexity of both the imperative variability and the declarative variability offered by PVDI. In order to explore their complexity further, let us first define a task of variability management as a non-empty set of possible variants $V_A = \{v_0, \dots\}$, each being a process made of activities of some finite set of activities A. Every variant represents a single legal modification of some business process, which is typically referred to as template process. We can then define the complexity of a given variability approach regarding a variability task V_A as the number of different structural variations or the number of constraints needed to describe this task as a function of the cardinality of the set A.

While considering imperative variability approaches, the complexity is directly related to the amount of structural variations needed in a template in order to express all possible variants. Thus, each variant $v \in V_A$ is the result of applying one or more structural variations. Therefore, the minimal number of structural variations is greater than or equal to the number of possible variants. Since the complexity equals the amount of structural variations, the complexity itself is also greater than or equal to the number of possible variants. Therefore, the minimal theoretical complexity equals 1 in the case of only one possible variant. The maximum theoretical complexity can be considered when there are no restrictions at all, that is, the set V_A contains all possible business processes which can be built based on the activities of



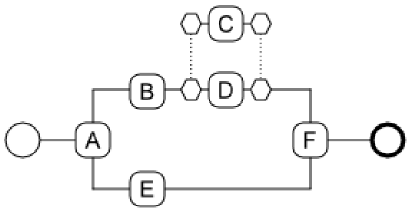
(a) Variant 1.



(b) Variant 2.

$$\begin{aligned}
&start \Rightarrow AX a\\
&a \Rightarrow EX b\\
&a \Rightarrow EX e\\
&a \Rightarrow AX(b \vee e)\\
&b \Rightarrow AX(c \vee d)\\
&c \Rightarrow AX f\\
&d \Rightarrow AX f\\
&e \Rightarrow AX f\\
&f \Rightarrow AX end
\end{aligned}$$

(c) Declarative Solution.



(d) Imperative Solution.

Figure 14: Two variants (a) and (b) and their declarative (c) and imperative (d) solutions.

the set A. To estimate that number, consider Theorem 9.4.

Theorem 9.4 (Theoretical complexity of imperative variability)**.** *The number of imperative structured variations for a business process build of N distinct activities and a finite number of gateways is at least $O(5^N)$.*

Proof. The maximal number of structured variations which do not involve the adding of removing of activities is equal to the number of non–equivalent business processes build of the same activities. This number we can use as a lower bound to estimate the number of possible structural variations.

For a given business process, built of activities $a_1, a_2, \dots a_N$, pick an arbitrary activity, for example, let it be a_1. Then, for each pair of activities a_1, a_k (where $2 \leq k \leq N$), the following options are possible:

1. Activity a_1 is always followed by activity a_k;
2. Activity a_1 is followed by activity a_k, but not always;
3. Activity a_k is always followed by activity a_1;
4. Activity a_k is followed by activity a_1, but not always;

5. Neither of the above statements is true, meaning that activities a_1 and a_k are parallel.

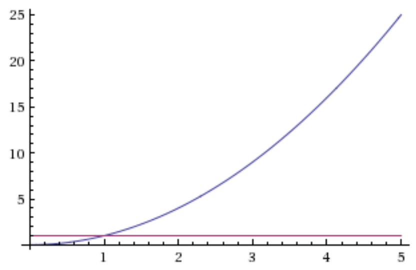
In total this results in $N-1$ pairs with 5 options per pair, and thus 5^{N-1} possible combinations. It must be noted that the number 5^{N-1} gives only the lower bound since we left out of consideration the possible relations between the other activities. As a result, the lower bound of the number of distinct business processes is of $O(5^N)$. $\square$

Let us consider the declarative-based variability used by PVDI. In this case, the complexity equals the number of constraints included in a template in order to restrict the number of possible variants. In the extreme case of high variability, when there are no restrictions at all, a PVDI template is quite simply a process. Which gives a constant complexity of 0. The other extreme occurs when no changes are allowed. Because it captures the complete process structure instead of single transitions, we use the frozen area Definition 4.9. A frozen area requires $2N+1$ formulas, each of length N, which gives in result $O(N^2)$ atomic formulas. Note that other low–variable cases can also be represented as a set of frozen or semi–frozen areas, which again results in a complexity of $O(N^2)$.

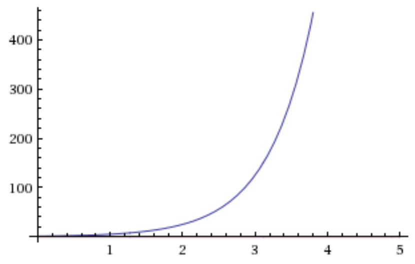
To summarize, in our proposed framework of comparison between imperative and declarative approaches, the relative complexity is constant versus $O(N^2)$ for the case of fixed process (Figure 15a) and $O(5^N)$ versus constant for the case of highly flexible process (Figure 15b). The immediate conclusion is that employing the PVDI approach significantly reduces the upper bound of the complexity, because this approach always results in a polynomial complexity whereas imperative ones range widely from constant to exponential complexity. Figure 15c illustrates this fact by depicting an indication of how the complexity of a template increases on the average for both variability techniques when capturing a low to high amount of variability.

10 Related Work

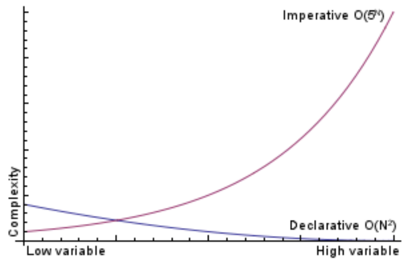
Several variability tools and frameworks have been proposed. In [3] we introduced the concepts and requirements of variability in BPM and classified existing frameworks and tools along those requirements. Table 4 gives an overview of the currently proposed frameworks and tools. Most of them



(a) Low variable case with constant (1) imperative and N^2 declarative complexity



(b) High variable case with constant (0) declarative and 5^N imperative complexity



(c) Both figures plotted against each other

Figure 15: Imperative versus declarative complexity.

can be classified as imperative frameworks and a smaller number as declarative ones. The majority concerns design–time variability where either during design–time a number of variations are created between which a consumer may choose prior to or directly before running the process, or the process is left underspecified to be completed by the end-user. Most of the design–time frameworks and tools focus on the design process itself and offer extensions to the modeling of business processes [6, 22–26]. Exceptions are [21, 27, 28] where extensions to the Business Process Execution Language (BPEL) offering variability through sets of variation points are proposed. Of these, [28] allows placeholders. The BPCN [19, 20] framework offers variability through an imperative process specification which includes so–called pockets of flexibility. These pockets are areas within the business process which are left underspecified to be modeled by a consumer prior to execution. ADEPT [29] and DECLARE [30] are tools which respectively offer imperative and declarative run–time variability. The latter of these includes a modeling language

	Design–time										Run–time		
Classification	**Chang et al.** [21]	**Gottschalk et al.** [22]	**Hadaytullah et al.** [23]	**Nguyen et al.** [24]	**Hallrtbach et al.** [25]	**Razavian et al.** [26]	**Sun et al.** [27]	**Mietzner et al.** [28]	**Lu et al.** [19]	**Groefsema et al.** [6]	**Dadam et al.** [29]	**Pesic et al.** [30]	**Marrella et al.** [31]
Pure imperative	√	√	√	√	√	√	√				√		
Imp. base with dec. techniques								√					
Imp. base with pockets of dec.									√				
Dec. base with imp. techniques										√			
Pure declarative												√	√
BPEL extension	√						√	√					
Modeling extension		√	√	√	√	√				√		√	

Table 4: Tools and frameworks classification.

called DecSerFlow [32] which is similar to the language of PVDI but focused on declarative techniques. DECLARE uses Linear Temporal Logic as its verification language, and as such is not able to formulate requirements of existence in a path as we do. However, as it is a run–time framework, only one path needs to be considered. In [31] a planning approach is used to adapt process models on the fly, while using a YAWL implementation to show feasibility. In the field of process support systems PROSYT [33] proposed an artifact based system where constraints are used to guard the artifacts but may sometimes be broken in favor of flexibility.

In [34] and [35] differences between the imperative and declarative approaches are studied from the design and maintainability perspectives. It was found that although the imperative approach is more understandible partly due to familiarity, both approaches have their merits and excell in different cases.

PVDI, therefore, employs a declarative approach to variability which, unlike other design–time frameworks, offers both declarative and imperative variability techniques. Because of its declarative foundation, its expressive power is bound by the lower bound in complexity. Whereas most other frameworks are bound by the higher imperative complexity. At the same time, declarative usability issues are ameliorated through a straightforward to use visual modeling extension from which the declarative constraint formulas can be generated directly.

11 Conclusion

Managing many variants of the same sort of processes is becoming a growing industrial need, especially to achieve mass customization and adaptation to several execution contexts. This has prompted for models and frameworks to deal with variability in an explicit manner in the field of business process management. Two techniques have emerged: imperative and declarative ones. Both are expressive and offer advantages for the process of engineering business processes.

Non-closed and non-finite templates however allow for more variability. Properties which declarative techniques possess naturally, and are sometimes included in imperative techniques through declarative mechanisms such as placeholders. However, these situations must be modeled explicitly during the design of the imperative template. Something which quickly complicates the design when dealing with a large amount of variations. Declarative approaches excel in very flexible situations, but have are an overkill to express straightforward enumerations of variants. Something which imperative techniques excel at. A natural conclusion is thus to choose the right approach on a per case basis. Though, this would require tool support for both techniques and possibly the necessity of migration from one tool to the other, with all the drawbacks that this implies.

A better option is to combine both techniques while limiting practical complexity. As discussed, imperative techniques have a complexity ranging between 1 for low variable templates to $O(5^N)$ for high variable templates. Declarative techniques on the other hand have a complexity ranging from $O(N^2)$ for low variable templates to 0 for high variable ones. When encoding one technique using the other, the lower declarative complexity is ultimately preferred. Our proposed PVDI does exactly this by employing a declarative approach with support for imperative methods encoded in large collections of declarative formulas and introducing a graphical layer which hide these large collections of formulas. By employing a graphical approach, PVDI allows a template designer to use a mix of both techniques while being able to avoid the complexity issues of both.

With PVDI we ameliorate a number of common BPM issues such as reusability and flexibility, as well as common BPM variability issues relating to complexity and usability. We do so by offering a graphical design extension to BPMN which translates the designer his wishes internally into constraints usable by the declarative foundation of PVDI. Through the addition of the graphical layer the complexity of the declarative formalism is completely hidden from the business process modeler. Additional graphical notations describing new behavior can be added easily to the extensible PVDI framework, offering an even larger range of options than described by this article. The foundation of PVDI is rooted in a declarative base, which we proved to be the lesser complex choice. Most existing frameworks however support the more complex and restrictive imperative variability. And those that support declarative mechanisms do so without much regard for the design process.

We developed tooling for PVDI in the form of VxBPMN and subjected it to performance tests using three different templates. Resulting from these tests was the fact that all three processes could be validated in between half a millisecond to 4 milliseconds with the slowest times per individual constraint for the process with the smallest process description. Thus, the validation mechanism scales well when dealing with larger process descriptions, but less when including complicated loop constructs.

Acknowledgements

The research is supported by the NWO Software as Services for Local eGovernments (SAS–LeG) project, http://www.sas-leg.net, contract No. 638.001.207. We thank the members of the SaS-LeG team for fruitful discussion, and especially we thank W. H. Hesselink for his valuable feedback on previous versions of the paper.

References

[1] W. van der Aalst and S. Jablonski, "Dealing with workflow change: Identification of issues and solutions," *International Journal of Computer Systems, Science, and Engineering*, vol. 15(5), pp. 267–276, 2000.

[2] M. Sinnema, S. Deelstra, and P. Hoekstra, "The COVAMOF Derivation Process," in *Int. Conf. On Software Reuse*, vol. LNCS. Springer, 2006, pp. 101–114.

[3] M. Aiello, P. Bulanov, and H. Groefsema, "Requirements and tools for variability management," in *IEEE workshop on Requirement Engineering for Services (REFS 2010) at IEEE COMPSAC*, 2010.

[4] C. Sun, R. Rossing, M. Sinnema, P. Bulanov, and M. Aiello, "Modelling and managing the variability of web service-based systems," *Journal of Systems and Software, Elsevier*, vol. 83, pp. 502–516, 2010.

[5] H. Schonenberg, R. Mans, N. Russell, N. Mulyar, and W. M. P. van der Aalst, "Process flexibility: A survey of contemporary approaches," in *CIAO! / EOMAS*, ser. LNBIP, J. L. G. Dietz, A. Albani, and J. Barjis, Eds., vol. 10. Springer, 2008, pp. 16–30.

[6] H. Groefsema, P. Bulanov, and M. Aiello, "Declarative enhancement framework for business processes," in *Int. Conference on Service-Oriented Computing, ICSOC, LNCS 7084*, 2011, pp. pp 495–504.

[7] N. Van Beest, P. Bulanov, J. Wortmann, and A. Lazovik, "Resolving business process interference via dynamic reconfiguration," in *8th International Conference on Service Oriented Computing (ICSOC-2010)*, vol. 6470/2010. Lecture Notes in Computer Science, 2010, pp. 47–60.

[8] ——, "Automated runtime repair of business processes," University of Groningen, Tech. Rep., 2012, http://www.cs.rug.nl/ẽirini/papers/tech_2012-12-2.pdf.

[9] E. A. Emerson and J. Y. Halpern, "Decision procedures and expressiveness in the temporal logic of branching time," in *Proceedings of the fourteenth annual ACM symposium on Theory of computing*, 1982, pp. 169–180.

[10] OMG, "Business process model and notation (bpmn) ftf beta 1 for version 2.0," OMG, Tech. Rep. dtc/2009-08-14, August 2009.

[11] E. A. Emerson and J. Y. Halpern, "Decision procedures and expressiveness in the temporal logic of branching time," *Journal of Computer and System Sciences*, vol. 30, no. 1, pp. 1–24, 1985.

[12] M. Wynn, H. Verbeek, W. van der Aalst, A. ter Hofstede, and D. Edmond, “Business process verification : finally a reality!” *Business Process Management Journal*, vol. 15, no. 1, pp. 74–92, 2009.

[13] E. M. Clarke, O. Grumberg, and D. A. Peled, *Model Checking.* MIT Press, 2000.

[14] E. M. Clarke, E. A. Emerson, and A. P. Sistla, “Automatic verification of finite-state concurrent systems using temporal logic specifications,” *ACM Transactions on Programming Languages and Systems*, vol. 8, no. 2, pp. 244–263, 1986.

[15] P. Bulanov, H. Groefsema, and M. Aiello, “Business process variability: A tool for declarative template design,” in *Int. Conference on Service-Oriented Computing - Demo Track (ICSOC–2011), LNCS 7221*, 2011, pp. 241–242.

[16] S. Balko, A. H. M. ter Hofstede, A. P. Barros, and M. L. Rosa, “Controlled flexibility and lifecycle management of business processes through extensibility,” in *EMISA*, 2009, pp. 97–110.

[17] B. Weber, M. Reichert, and S. Rinderle-Ma, “Change patterns and change support features - enhancing flexibility in process-aware information systems,” *Data Knowl. Eng.*, vol. 66, no. 3, pp. 438–466, 2008.

[18] R. Milner, *Communication and concurrency.* Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 1989.

[19] R. Lu, S. Sadiq, and G. Governatori, “On managing business processes variants,” *Data Knowl. Eng.*, vol. 68, no. 7, pp. 642–664, 2009.

[20] S. W. Sadiq, M. E. Orlowska, and W. Sadiq, “Specification and validation of process constraints for flexible workflows,” *Inf. Syst.*, vol. 30, no. 5, pp. 349–378, 2005.

[21] S. H. Chang and S. D. Kim, “A variability modeling method for adaptable services in service-oriented computing,” in *11th International Software Product Line Conference, 2007. SPLC 2007.*, 2007, pp. pp 261–268.

[22] F. Gottschalk, W. M. P. van der Aalst, M. H. Jansen-Vullers, and M. L. Rosa, “Configurable workflow models,” *Int. J. Cooperative Inf. Syst.*, vol. 17, no. 2, pp. 177–221, 2008.

[23] K. Hadaytullah Koskimies and T. Systa, "Using model customization for variability management in service compositions," in *IEEE International Conference on Web Services, 2009. ICWS 2009.*, 2009, pp. pp 687–694.

[24] T. Nguyen, A. Colman, and J. Han, "Modeling and managing variability in process-based service compositions," in *Service-Oriented Computing*, ser. Lecture Notes in Computer Science. Springer Berlin / Heidelberg, 2011, vol. 7084, pp. 404–420.

[25] A. Hallerbach, T. Bauer, and M. Reichert, "Managing process variants in the process life cycle," in *ICEIS (3-2)*, 2008, pp. 154–161.

[26] M. Razavian and R. Khosravi, "Modeling variability in business process models using uml," in *Fifth International Conference on Information Technology: New Generations, 2008. ITNG 2008.*, 2008, pp. pp 82–87.

[27] C. Sun and M. Aiello, "Towards variable service compositions using VxBPEL," in *Int. Conf. On Software Reuse*, vol. LNCS 5030. Springer, 2008, pp. 257–261.

[28] R. Mietzner and F. Leymann, "Generation of bpel customization processes for saas applications from variability descriptors," in *2008 IEEE International Conference on Services Computing Vol. 2*, 2008, pp. pp 359–366.

[29] P. Dadam and M. Reichert, "The adept project: a decade of research and development for robust and flexible process support," *Computer Science - R&D*, vol. 23, no. 2, pp. 81–97, 2009.

[30] M. Pesic, M. H. Schonenberg, N. Sidorova, and W. M. P. van der Aalst, "Constraint-based workflow models: Change made easy," in *OTM Conferences (1)*, 2007, pp. 77–94.

[31] A. Marrella, M. Mecella, and A. Russo, "Featuring automatic adaptivity through workflow enactment and planning," in *7th International Conference on Collaborative Computing: Networking, Applications and Worksharing (CollaborateCom 2011)*, 2011.

[32] W. van der Aalst and M. Pesic, "Decserflow: Towards a truly declarative service flow language," in *Web Services and Formal Methods*, ser. Lecture Notes in Computer Science. Springer Berlin / Heidelberg, 2006, vol. 4184, pp. 1–23.

[33] G. Cugola, "Tolerating deviations in process support systems via flexible enactment of process models," *IEEE Trans. Softw. Eng.*, vol. 24, no. 11, pp. 982–1001, Nov. 1998.

[34] P. Pichler, B. Weber, S. Zugal, J. Pinggera, J. Mendling, and H. A. Reijers, "Imperative versus declarative process modeling languages: An empirical investigation," in *Business Process Management Workshops*, ser. Lecture Notes in Business Information Processing, F. Daniel, K. Barkaoui, S. Dustdar, W. Aalst, J. Mylopoulos, M. Rosemann, M. J. Shaw, and C. Szyperski, Eds. Springer Berlin Heidelberg, 2012, vol. 99, pp. 383–394.

[35] D. Fahland, J. Mendling, H. Reijers, B. Weber, M. Weidlich, and S. Zugal, "Declarative vs. Imperative Process Modeling Languages: The Issue of Maintainability," in *1st International Workshop on Empirical Research in Business Process Management (ER-BPM'09)*, B. Mutschler, R. Wieringa, and J. Recker, Eds., Ulm, Germany, Sep. 2009, pp. 65–76, (LNBIP to appear).

A Computational Tree Logic in a Nutshell

Computational Tree Logic (CTL) and Computational Tree Logic$^+$ (CTL$^+$) are Branching–time Logics and are defined inductively by applying the following rules 1-5 and 1-6 respectively [9,11]. Although CTL^+ is equivalent to CTL, it does allow for more compact and understandable formulas. A CTL *model* $\mathcal{M} = \langle S, T, L \rangle$ consists of a set of states S, a set of transitions T, and a valuation function L. And a *path* is a sequence $(s_0, s_1, ...)$ of states such that $(s_i, s_{i+1}) \in T$.

1. Each primitive formula is a state formula
2. If p, q are state formulas, then so are $(p \wedge q)$ and $\neg p$
3. If p is a state formula, then Xp, Fp and Gp are path formulas, with
   - Next: Xp meaning that the path has a ne$\underline{\text{X}}$t state, and there p holds
   - Finally: Fp meaning that at some state, eventually, p holds

   - Globally: Gp meaning that Globally, at all states, p holds

4. If p is a path formula, then Ep and Ap are state formulas, with
   - Exist: Ep meaning that there Exists a path for which p holds
   - Always: Ap meaning that All paths satisfy p
5. If p, q are state formulas, then $[p\ U\ q]$ and $[p\ W\ q]$ are path formulas, with
   - Until: $[p\ U\ q]$ meaning that at some future state in the path q holds, and Until that point all states satisfy p
   - Weak until: $[p\ W\ q]$ meaning that unless q holds in a state, p holds up until that state or globally otherwise
6. If p, q are path formulas, then so are $p \wedge q$ and $\neg p$