Business process variants:
A generation from templates

Robbert-Jan Pijpker

university of groningen

faculty of mathematics and natural sciences

Daily Supervisor: Drs. H. Groefsema
First Supervisor: Prof. dr. ir. M. Aiello
Second Supervisor: Prof. dr. G.R. Renardel de Lavalette

Computing Science
Faculty of Mathematics and Natural Sciences
University of Groningen
Groningen

Abstract

Variability in Business Process Management is becoming increasingly important. To deal with this, Aiello et al. introduced a technique called PVDI that offers both declarative and imperative variability. This technique has been presented in a variability tool named VxBPMN.

We have designed and implemented a way to generate the simplest variant of a template and we have extended VxBPMN with this implementation. Generating the simplest variant of a template is done by creating all possible paths of the template and removing some of them by using the validation method of the VxBPMN. We will explain how the generator works and we will show some results of the tests that we have run.

Contents

Chapter 1

Introduction

A business process is a collection of several activities that results in a product or service for a particular customer or customers. An example of a business process is shown in Figure 1.1. This example shows the process of reserving a classroom. The activities here are choosing a practical room, locking the system, viewing which rooms are free and reserving the room. The result of this business process will be a reserved practical room. Business Process Management, BPM, is about managing business processes and focuses mainly on efficiency and effectiveness[8].

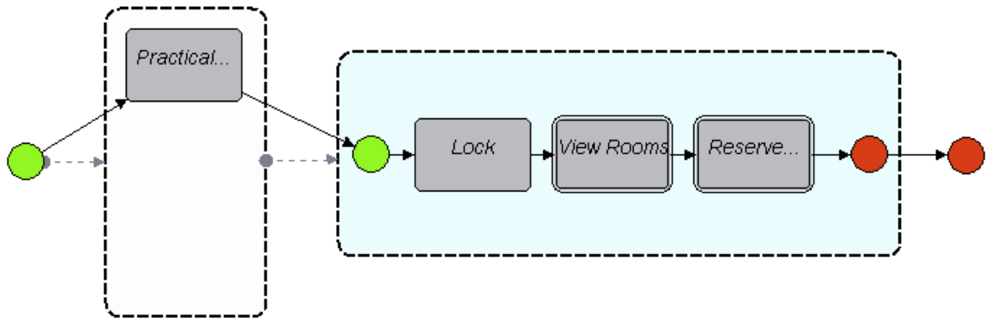


Figure 1.1: An example of a business process. *Source: [4]*

1.1 Variability

With the increasing improvements in customization of business processes, the role of variability in business processes is becoming increasingly important in Business Process Management. Variability offers re-usability and flexibility which increases the effectiveness of a process. In the past BPM did not support dynamically changing workflows and thus did not support any variability[7]. Variability in the domain of BPM means that not everything in a process is static. Certain parts of it remain open for change or are not fully defined. This leads to more than one variant of the same business process, based on its intended use. Aiello et al. have introduced in [3] a technique that introduces variability to the domain of BPM, called PVDI. PVDI offers variability in a business process through

Declarative and Imperative techniques. Declarative techniques and Imperative techniques are both flexibility techniques. Declarative techniques allow every variation, except for the variation that is specifically disallowed and Imperative techniques only allow variations that are strictly specified. By combining these techniques, a template designer can use a mix of both and can thus avoid the disadvantages of both techniques.

A process in PVDI is defined as a directed graph that consists out of activities, gates and events. In more formal language this is:

Definition 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;*
- *$A = \{A_1 ... A_n\}$ is a finite set of activities;*
- *$G = G_a \cup G_x$ is a finite set gateways, consisting of and- and xor-gates as defined by BPMN;*
- *E is a set of events, containing a unique start $\odot$ and $\otimes$ end 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 $(a, s) \in T$;*
- *$\forall s \in S \backslash \{\otimes\}$ there is at least one $s' \in S$ with $(s, s') \in T$.*

The PVDI framework makes it possible to introduce constraints for processes and since a template simply can be seen as a process with constraints, it is now possible to make a template. The constraints are made by using computational tree logic, CTL^+. CTL^+ is a Branching-time Logic. A CTL^+ model $\mathcal{M} = \langle S, T, L \rangle$ is build up out of a set of states S, a set of transitions T, and a valuation function L. A path is a sequence $(s_0, s_1, ...)$ of states for which $(s_i, s_{i+1}) \in T$ [1].

CTL^+ is defined by the following rules:

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 which say that either the path p holds on the next state in the path, on some state in the path or that at all states p holds
4. If p is a path formula then Ep is a state formula which says that some path satisfies p
5. If p is a path formula then Ap is a state formula which says that all paths satisfy p
6. If p, q are state formulas then $p \cup q$ is a path formula which says that there is some state on the path that satisfies q, and all states before it satisfy p
7. If p, q are path formulas, then so are $p \wedge q$ and $\neg p$

The definition of a constraint is:

Definition 2 (Constraint) *A constraint ϕ over a process P is a CTL^+ formula whose propositional variables are in $L(S)$ of P, where L is a labeling function using the natural valuation.*

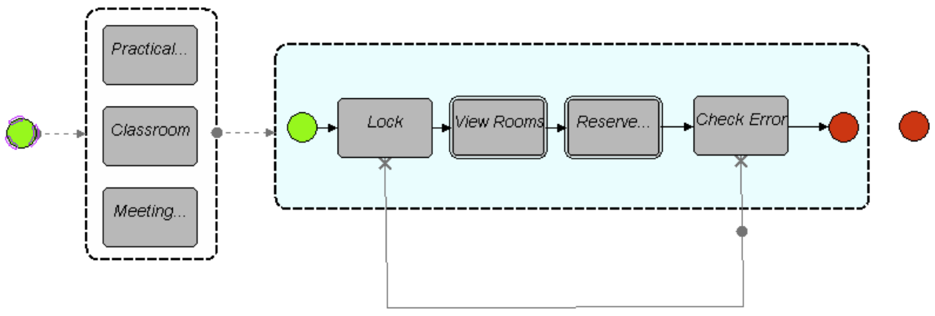


Figure 1.2: PVDI template for room reservation. *Source:[4]*

To be able to use constraints and thus evaluate a process, a set of variables and a valuation function is introduced. For every node, every constraint should be evaluated to TRUE. This valuation function is defined as:

Definition 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 $\forall s \in S : \mathcal{M}, s \models \phi$.*

As mentioned before, a template can be considered a process with constraints. This means that Definition 1 and Definition 2 can be combined into a definition for a template:

Definition 4 (Template) *A template R is a tuple $\langle S, T, \phi \rangle$*

- *S as defined in Definition 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 2.*

A variant of a template is a process for which all the constraints of the template are valid. For example Figure 1.1 shows a variant for the template shown in Figure 1.2.

Definition 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.*

Aiello et al. introduced a group in PVDI to describe constraints that span over a set of nodes. The definition of this is:

Definition 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^+ formulate are replaced by $(s_1 \vee ... \vee s_n)$ where $s_1 ... s_n$ are elements of the group s_g.*

1.2 BPMN

For modeling of a business process, Business Process Model and Notation, BPMN, can be used. BPMN provides a graphical notation of a business process and can be used to drawing business processes in a workflow[6]. PVDI expands this notation with PVDI template elements. Each of these elements can be split up in two parts: The graphical part and the CTL^+ part. The graphical parts are shown in Figure 1.3 while the CTL^+ parts are used to make a constraint. We will shortly describe the elements A to D and ordered execution here. The other elements are not relevant for this thesis and are thus not explained here. All of these can be found in [4].

A: Mandatory Execution Constraint For a node with a mandatory execution constraint, all paths should contain this node.

B: Mandatory Selection Constraint For a node with a mandatory selection constraint, there should be at least one path that contains the node.

C: Normal/Optional A node that is optional does not have to be in the business process and it is thus allowed to remove such a node.

D: Frozen area A frozen area is a constraint on a part of a template for which all the nodes in the area become mandatory and every path in the area allows for no variation.

Ordered Execution Constraint An ordered execution constraint is a relation between two nodes $p \in S$ and $q \in S$ and can be split up into two dimensions. Namely the path and the distance. The distance dimension can be split up in F, Finally and X, neXt, which means that nodes should either eventually or immediately follow each other. The path dimension can be split up in E, Exists, and A, All, which means that the nodes follow each other in either one path or all path. The graphical representation of this is shown at the bottom of Figure 1.3.

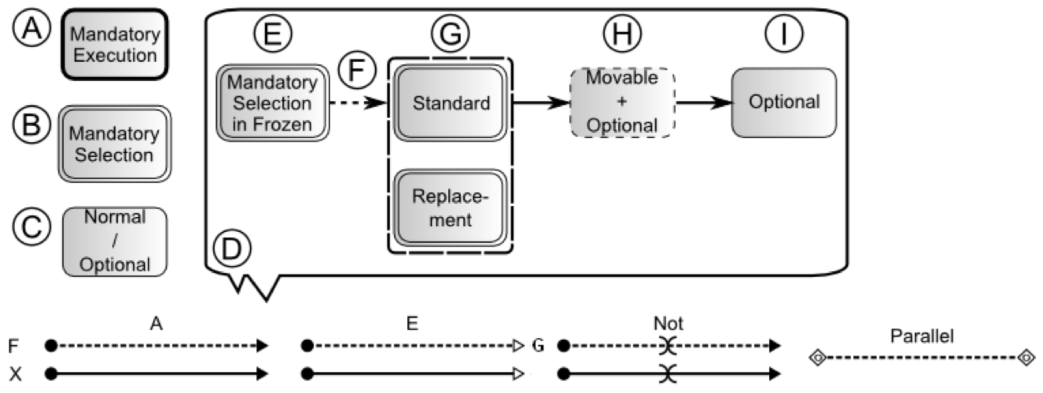


Figure 1.3: PVDI elements. *Source: [4]*

1.3 Overview

The goal of this thesis is to design and implement an algorithm to find the simplest business process variant of a template. We have extended the VxBPMN tool, that is a PVDI tool introduced by [4], with this algorithm. Chapter 2 will explain about the requirements of the thesis. In Chapter 3, we mention which approaches can be used to generate a business process and which approach we have used. Chapter 4 shows and explains how this has been realized. In Chapter 5, the results of the research are discussed and in Chapter 7 we mention work that is related to this thesis. Finally, in Chapter 6, some conclusions are drawn about the approach that we have used.

Chapter 2

Requirements

When we started to work on this thesis, the original project existed out of two parts. These were:

- Design and implement an algorithm which finds and saves all possible correct variants of a given template.
- Design and implement an algorithm which finds and saves the simplest, smallest or computational cheapest correct variant of a given template.

Because of the fact that the project was initially made for one to two people, the project has been reduced to just the second part: Design and implement an algorithm which finds and saves the simplest, smallest or computational cheapest correct variant of a given template.

This left us with the choice which option we would choose: the simplest, smallest or computational cheapest. We have chosen to implement an algorithm that finds the correct variant that is the simplest. The simplest variant is the variant that has the least paths from start to end and should contain the least amount of loops possible. We have chosen to implement an algorithm to find the simplest variant because we believe that the simplest is also often small and probably also computational cheap.

The requirements for the project are now:

1. The found variant should contain the least amount of paths from start to end as possible;
2. The found variant should contain the least amount of loops possible;

Chapter 3

Approach

To generate a valid business process from a template, you have to keep in mind what the definition is of a valid business process. A valid business process, is a business process that is healthy and for which all the constraints are valid[4]. Since a path from start to end is also a variant, a path can be evaluated. When this is evaluated to false, the path is in conflict with at least one of the constraints and should thus be removed from the process. With this in mind, we have been looking at several approaches to generate a valid variant.

3.1 CTL

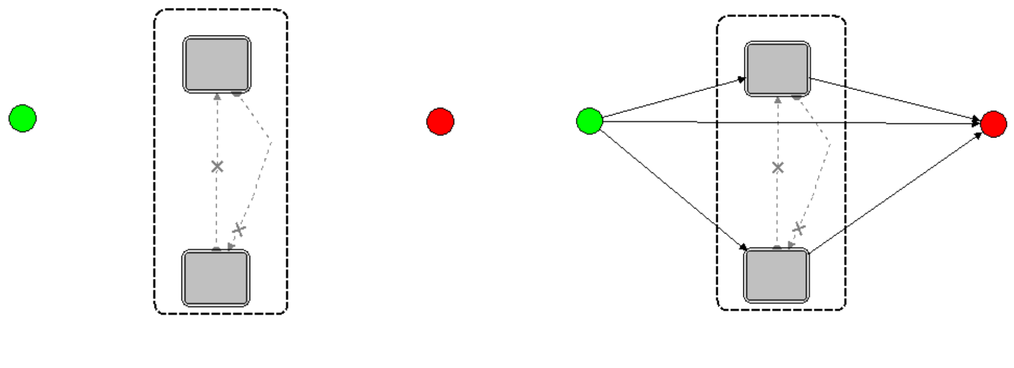
We have already explained that all the constraints are made of CTL^+. This means, that it should be possible to generate a variant from this CTL^+. A valid variant means that all the constraints should be evaluated to TRUE. Thus, to generate a valid variant from the CTL^+, all the formulas of the template have to be solved.

3.2 One Valid Path

Since a path from start to end is also a variant and should also be validated, all the paths have to be generated and validated. However, a found path that is valid is also a valid variant. For example the variant shown in Figure 1.1 is a path from start to end that is not in conflict with any of the constraints, and is thus a valid path. When a valid path is found, this path can be saved and used as a variant and the generation can stop. This approach is the fastest way to find a variant, but might not always result in one. For example, the business process shown in 3.1b is a valid variant of the template shown in Figure 3.1a, but none of its paths are valid.

3.3 Generate all possibilities

The only way to be entirely certain that the generation of a business process from a template will result in a valid business process when there is a possible valid business process, is by trying all the possible variants of a template. To do this, all the possible paths from the start to the end have to be evaluated. When the path is in conflict with any of the constraints of the template, a flow from this path will be deleted and the generation continues. This will eventually result in either a valid business process or an invalid path from which no flows can be removed. The first case means that a valid variant has been found and that the generation is done, but in the second case no valid variant has been found. Ending up with an invalid path from which no flows are allowed to be removed can mean two things: Either there is no valid variant of the template, or a wrong flow has been removed during the generation, and thus some backtracking has to be done.



(a) Example template (b) Example variant

Figure 3.1: A template and variant that have no valid paths

3.4 Pre-processing

Since we are searching for the simplest variant of a template, all the optional parts of a template have to be removed. Also, since all paths have to be generated and evaluated, the amount of paths has the biggest influence on the execution time of the generation. This is why some pre-processing has to be done. All the optional nodes of a template have to be removed and the generation can be sped up by having a look at ordered execution.

3.4.1 Deleting Nodes

For the generation of paths, all the optional nodes of a template will result in optional paths. Since we are searching for the simplest variant of a template, all these optional paths have to be removed. A way to do this, is by removing all the optional nodes of a template. Every node that does not have any constraints is optional and can thus be deleted while making a business process. These nodes will eventually have to be removed since they are not part of the simplest variant and by doing this at the beginning, all the paths that would otherwise contain this node are now never generated and thus the amount of paths decreases. This can really speed up the generation, especially with larger templates.

3.4.2 Ordered Execution

Some flows can be left out of the generation when you take ordered execution in consideration. With the earlier given definition in mind, we can have a look at three of the ordered execution relations:

- For-All-Next: For the relation that says for All the paths from p, the Next node is q, the only flow that goes somewhere from p is towards q. All the other flows going from p will result in invalid paths and will be eventually deleted so these flows can be left out of the generation in the first place.
- For-All-Not-Next: This relation indicates that for All the paths from p, the Next node is not allowed to be q. This means that it is not allowed to have a flow that connects p with q.
- For-All-Not-Finally: In All the paths from p, the node q is never allowed to be in the path. This means that we can do the same as with the For-All-Not-Next relation. We can just leave out the flow between p and q.

By leaving out the flows in the way described above, some paths will be left out of the generation. These are paths that would later on be removed because they are in conflict with some of the constraints. By not generating them in the first the place, the generation of the simplest variant is sped up.

3.5 Chosen approaches

In our implementation, we have left out the CTL^+ approach since in order to solve every CTL^+ formula, also all the sub-formulas have to be solved and this requires a lot of computations. Instead, we have chosen to combine all of the other approaches.

Chapter 4

Realization

In the previous chapter, we have explained about the different approaches for generating a valid variant of a template. In this chapter we will explain about how we have implemented these approaches.

4.1 Generating Flows

First of all, all the flows between all the nodes have to be made. Since constraints on groups work different than constraints on nodes, groups should be treated in a different way. This why we have chosen to first make all the flows between all the nodes and make the flows within a group afterwards. The generation of flows is done by iterating over all the nodes, checking if it is allowed to add a flow between them and if so, adding the flow. The check if it is allowed to make a flow is part of the pre-processing of ordered execution that we discussed in the previous chapter.

For actually making the flow, it is important to know of what instance the source and target node are. For example, if the source node is an instance of a frozen group, you do not want a flow going directly from the frozen group towards the target node since outgoing flows from the frozen group are only allowed to have the end of the frozen group as a source. This is why you want the source node to be the end of the frozen group. The other way around gives the same situation, the target node should not be the frozen group, but the start of the frozen group, since frozen groups only allow the start to have incoming flows from outside of the frozen group. To make a flow towards a group, you have to make a flow towards all the nodes inside the group. The other way around, a flow has to be made from all the nodes in the group towards the target node. This is because of the way the constraints of a group work.

After all these flows have been made, the flows inside the groups have to be made. The flows inside a frozen group are part of the template and have thus already been made. This means that only normal groups remain and to make these flows, the same as above described is done again, but now with a different set of nodes, namely, the nodes of the group.

4.2 Finding paths

There are two different ways for finding all the paths: Depth First Search (DFS) and Breadth First Search (BFS). The difference between these two is that BFS will always first find the shortest paths while this is not the case for DFS[2]. That is the reason why we have chosen to implement BFS. Consider a short, invalid path p. A flow f has to be removed from p. The chance of removing the wrong flow is small because p is a short path. Deleting a flow from p does not only mean that you delete p from the process, but also all the paths that p is a sub-path of. By starting to remove smaller paths first, a lot of paths are immediately deleted and the chance that these should not be deleted is small.

4.3 Path validation

For checking if a path is valid, we are using the validation method as mentioned in [4]. This validation method checks for every constraint of the template if a node in the path is in conflict with it. When there is no node that is in conflict with any constraint, no flow will be removed and the path will remain a part of the process. When, however, there is a node that is in conflict with a constraint, this path is not valid and thus it has to be removed from the process. As mentioned before, removing a path from a process is done by removing a flow of the path from the process.

We have chosen to use a First Come First Served idea for removing flows. The first flow that is allowed to be removed from an invalid path, will be removed. We keep track of an error count of the process to check if flow is allowed to be removed. Before removing the flow, the amount of errors in the process is counted. This is done again after the flow has been removed and if the amount of errors has increased, the wrong flow has been removed. In this case, the flow is placed back and the next flow will be checked for removal. This is done until a flow is found that can be removed, or until all the flows in the path have been checked.

4.4 Backtracking

When every flow of an invalid path is not allowed to be removed, the path itself cannot be removed. This can mean two things: Either there is no valid business process for the template or a flow from a previous path has been removed that should not have been removed. In the first case the generation is done, but in the second case some actions have to be made undone. To do this, we have implemented a backtracking algorithm by using recursion, as shown in Listing 1. The idea behind this is as follows: All the paths are stored in a list sorted from small to large because of the BFS. When a path is invalid, a flow will be removed and the next path is called recursively. When all the paths are checked, the validation of the process will be returned. The previous path receives this return value and when this value is true, it returns it as well and the generation will end with a valid business process. If the value is false, a different flow from the path will be tried to remove and the next path will be called again.

Listing 1: checkNextPath

```
private boolean checkNextPath(int idx)
{
  if(idx < allPaths.size())
  {
    boolean pathDone = false;
    Path path = allPaths.get(idx);
    PathChecker pathChecker = new PathChecker(path, template);

    if(!path.containsFlow(deletedFlows)
      && pathChecker.checkError())
    {

      int errorCount = template.getErrorCount();
      Flow flow = null;

      path.getForbiddenFlows().clear();
      while(!pathDone
        && (flow = pathChecker.flowToRemove()) != null)
      {
        deletedFlows.add(flow);
        path.addForbiddenFlow(flow);
        pathDone = checkNextPath(idx+1);

        if(!pathDone)
        {
          template.setErrorCount(errorCount);
          deletedFlows.remove(flow);
          flow.getSource().addFlow(flow);
        }

      }
      return pathDone;
    }
    else
    {
      return checkNextPath(idx+1);
    }
  }

  return template.checkProcess();
}
```

4.5 Post-processing

To end the generation of a business process, some post-processing has to be done. We have chosen to split this into two parts, namely: making the business process healthy and removing loops in the business process.

4.5.1 Healthiness

As mentioned in [9], a business process should be healthy. One aspect of a healthy business process is that it does not contain any dead ends. After the deletion of some flows, it might be possible that there are nodes that do not have any incoming flows anymore. Such a node is completely disconnected from the entire business process and will thus violate the property of a healthy business process that it should not contain any dead ends. To solve this, the node will simply be removed. This can be done without creating any problems with any of the constraints because the fact that the node was completely disconnected from the rest, already means that it is not a part of the business process.

4.5.2 Removing loops

Since we are searching for the simplest variant of a template, the variant should be simplified as much as possible. This is why we remove all the loops that can be removed. The removal of a loop is done by a DFS, keeping track of all the nodes that have already been traversed. Whenever a node is found that has already been traversed before, the flow towards this node is removed and the business process will be evaluated. If the business process is now invalid, the flow that has been removed should not have been removed and is placed back. This way every loop that can be removed, will be removed.

Chapter 5

Results

We have included some tests to illustrate the result of the generation. Consider the template as shown in Figure 5.1. Two possible variants of this template are shown in Figure 5.2. Variant 5.2a is the variant that the generator finds by checking if a path is already a variant, as mention in Chapter 3. We have disabled this function to get the variant as shown in 5.2b. We have done this in order to explain how the backtracking that we have implemented works.

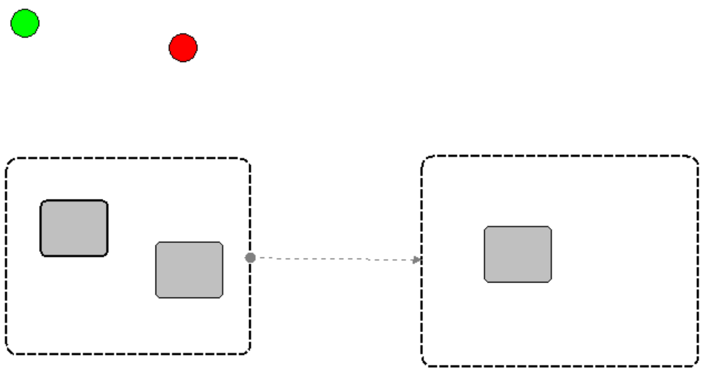
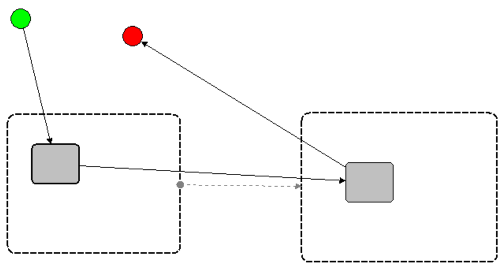
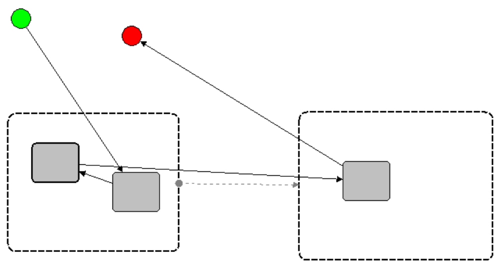


Figure 5.1: An example template for a business process



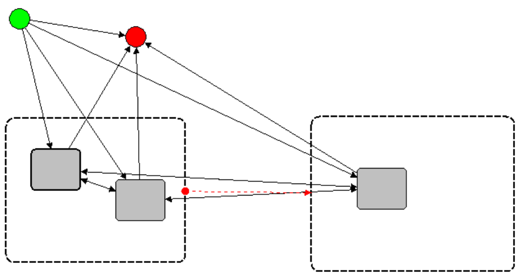
(a) Found by checking if a path is also a variant

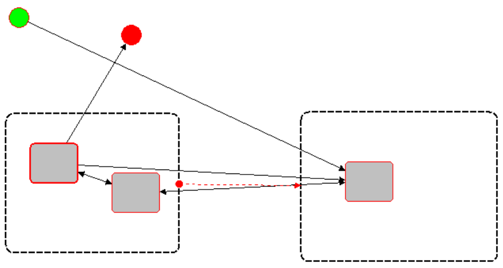


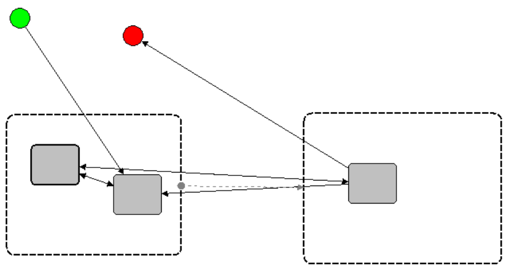
(b) Found by backtracking

Figure 5.2: Two possible variants of the template shown in 5.1

First of all, all the flows between all the nodes will be made, as shown in 5.3a. From here on, paths will be evaluated and flows will be removed for invalid paths. Doing this eventually results in the variant as shown in 5.3b. From here on, the generator will start backtracking its steps and finds the variant 5.3c. This variant is evaluated as valid and thus will the generator try to remove as many loops as possible. For this example, all the loops can be removed and variant 5.3d is finally found.


(a) All flows


(b) A wrong variant


(c) The variant found after backtracking

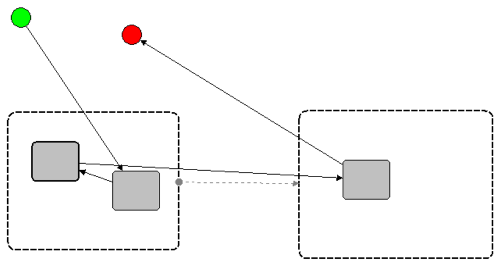

(d) The result after removing loops

Figure 5.3: The stages of backtracking

These tests show that both the approaches of finding a business process are working correctly and that they will, if possible, result in a valid business process.

Chapter 6

Conclusions

There are several approaches for generating a valid variant of a template. These approaches can be to find a path in a template that is also a valid variant, to generate all variants and use the first one that is valid, or to have a look at the CTL^+ behind the template. From these three approaches, the third requires the most computations. We have discarded this approach because we believe it is computational more expensive than the other approaches. This leaves us with the other two approaches. Finding a path that is also a valid variant of a template is the quickest way to find a variant, but might not always result in one since a valid variant can exists out of more than one path. Generating all possible variants and using the first valid variant that is found always results in a valid variant if the template is correct, but is quite slow because all the possible paths have to be generated first. This is why combining the two approaches is the best option. If there is a path that is also a valid variant, the generation will immediately result in this variant while if there is no such path, all the possibilities will be considered and the generation will eventually result in a valid variant.

The variant that is found is the simplest variant since it does not violate any of the three requirements. By ordering the paths from shortest first to largest last, all the shortest paths and thus also the shortest variants are first considered. This way, the found variant always has the least amount of nodes possible. Also, by deleting all the paths that are in conflict with any of the constraint, the only paths that are left are necessary for the variant to be valid and thus the variant contains the least amount of paths from start to end. Finally, by removing all the loops that can be removed, the variant contains the least amount of loops possible.

Chapter 7

Related Work

The related work of this thesis can be split up in three different parts: VxBPMN, generation of processes and graph theory.

7.1 VxBPMN

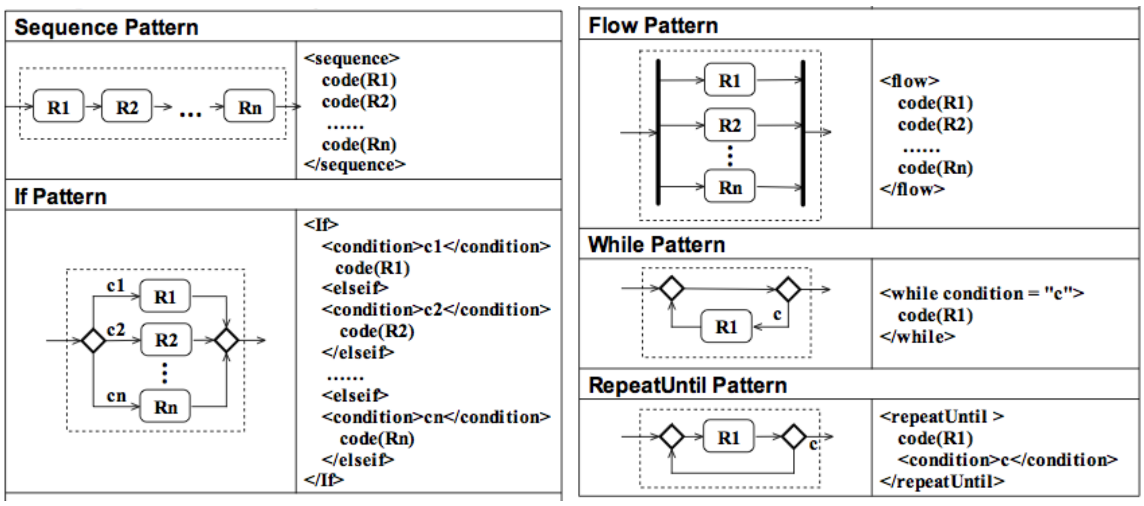
This thesis is based on the research of Aiello et al. [4]. Aiello et al. have introduced a technique that allows a combination of declarative and imperative techniques and have created a developing tool in the form of VxBPMN. This tool can be used to develop a template and validate a variant. We have extended the tool with our implementation of a business process generator.

7.2 Generation of processes

To our knowledge, no one else has published something about generating a business process from a template. We did find some different techniques for generating a process and we will discuss these here, but they all differ from our thesis on the point that all the other techniques need some sort of input in order to generate a process.

7.2.1 Meitzner and Leymann

Business Process Execution Language, BPEL, offers variability through sets of variation points. Mietzner and Leymann [5] present an approach that can generate BPEL customization processes out of variability descriptors. A variability descriptor consists out of a list of variability points and dependencies between the variability points. A variability point specifies the point in a software design that can be customized and thus offers variability, and a dependency describes a relation between variability points. A BPEL Process Model can then be made from the variability descriptors called a customization process. This process is used to prompt a user for inputs to bind the variability points in the order that can be retrieved from the dependencies. To generate BPEL, first activities are generated from the variability points that are not the target of a dependency. Afterwards, the next activity is generated by using the dependencies of the variability points. This finally results in a customization process.



Sequence Pattern

```
<sequence>
  code(R1)
  code(R2)
  ......
  code(Rn)
</sequence>
```

If Pattern

```
<If>
  <condition>c1</condition>
    code(R1)
  <elseif>
  <condition>c2</condition>
    code(R2)
  </elseif>
  ......
  <elseif>
  <condition>cn</condition>
    code(Rn)
  </elseif>
</If>
```

Flow Pattern

```
<flow>
  code(R1)
  code(R2)
  ......
  code(Rn)
</flow>
```

While Pattern

```
<while condition = "c">
  code(R1)
</while>
```

RepeatUntil Pattern

```
<repeatUntil >
  code(R1)
  <condition>c</condition>
</repeatUntil>
```

Figure 7.1: Example of transformation from UML to BPEL *Source: [10]*

7.2.2 Zhao, Duan and Zhang

Zhao, Duan and Zhang [10] introduce a UML 2.0 based approach for business process modeling and a set of transformation patterns to generate BPEL process models from this. Every activity is defined in the UML as a UML node and can be transformed to a BPEL activity. An example of how this can be done is shown in Figure 7.1.

7.3 Graph Theory

A business process can be modeled as a directed graph by using BPMN which means that most of the theorems about directed graphs are applicable for this project. Based on this knowledge, we can use different algorithms of graph traversal for finding and validating paths.

Chapter 8

Bibliography

[1] E. Allen Emerson and Joseph 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*, STOC '82, pages 169–180, New York, NY, USA, 1982. ACM.

[2] Michael T. Goodrich and Roberto Tamassia. *Algorithm Design: Foundations, Analysis and Internet Examples.* John Wiley & Sons, Inc., New York, NY, USA, 2nd edition, 2009.

[3] Heerko Groefsema, Pavel Bulanov, and Marco Aiello. Declarative enhancement framework for business processes. In Gerti Kappel, Zakaria Maamar, and Hamid R. Motahari Nezhad, editors, *ICSOC*, volume 7084 of *Lecture Notes in Computer Science*, pages 495–504. Springer, 2011.

[4] Marco Aiello Heerko Groefsema, Pavel Bulanov. Imperative versus Declarative Process Variability: Why Choose? January 2013.

[5] R. Mietzner and F. Leymann. Generation of bpel customization processes for saas applications from variability descriptors. In *Services Computing, 2008. SCC '08. IEEE International Conference on*, volume 2, pages 359–366, 2008.

[6] OMG. Business process model and notation (bpmn) version 2.0. OMG Documentation formal/2011-01-03.

[7] W. M. P. van der Aalst and S. Jablonski. Dealing with workflow change: identification of issues and solutions. *International Journal of Computer Systems Science and Engineering*, 15(5):267–276, September 2000.

[8] Mathias Weske. Business process management concepts, languages, architectures. Hardcover, November 2007.

[9] M. T. Wynn, H. M. W. Verbeek, Aalst, A. H. M. Ter Hofstede, and D. Edmond. Business process verification finally a reality! *Business Process Management Journal*, 15(1):74–92, 2009.

[10] Chenting Zhao, Zhenhua Duan, and Man Zhang. A model-driven approach for generating business processes and process interaction semantics. In *Computer and Information Science, 2009. ICIS 2009. Eighth IEEE/ACIS International Conference on*, pages 483–488, 2009.