Using BPMN-Q to show violation of execution ordering compliance rules

Manuel Blechschmidt manuel.blechschmidt@student.hpi.uni-potsdam.de

Hasso-Plattner-Institut
Prof.-Dr.-Helmer-Str 2-3
14482 Potsdam

Abstract. In the current world a lot of businesses are willing to comply to certain standards and regulations running their services. Normally these companies have a large amount of business processes, which have to fulfill the requirements, which are given by the law, standards or own policies. To run their business efficient, they are using process models which are complex and changing frequently. There is a big demand for automatic compliance checking against these repositories and showing of the violations, which were found during the test. At the moment these tests are expensive in salary and computing time and their output is not easy understandable. There is a need for them to be cheap in performance, repeatable and in parallel generating an output format, which the business architect can understand. This paper will introduce an approach how to use a BPMN-Q[1] pattern to show the compliance of a given model by generating an anti-pattern and using it to find a counter example. The counter example can be used for a visual representation of the violation.

BPMN-Q, compliance checking, model checking, ordering rules

1 Introduction

Today companies are using businesses processes and business process models to run their services smoothly. The artifacts are defining how the enterprise works and they are a good way of controlling, if the company is still following there goals. These models are kept in repositories. As for every sophisticated software system, this database needs to be maintaining by skilled engineers. There is a need for these processes to apply to certain standards or laws. Partly because they are given by the government and sometimes to have an advantage against a competitor. One of these laws would be the Sarbanes-Oxley Act for the banking industry. An other standard, which would give the company the chance to go ahead of another player, would be the ISO 9000 standard family, which ensures the quality processes in a company.

If a company wants to proof, that they apply to these specifications, they have to go through every process manually and show that this process does validate against the rules. The cost for compliance experts are high because, there are just a few of them, it takes them a long time to go through all models and the salary is increasing every year[6]. At the moment it seems that this situation will not change in the near future. [7]

The enterprises want to automate the compliance checking process, make it as fast as necessary, as easy as possible and have the possibility to run it over and over again always when there processes are changing.

Current model checkers have an expensive approach to first transform the processes to their own internal format [2] [4] and then run the given checks against them. When they find a counter example, they show an internal trace, which describes the found problem. Most of the time this internal tracing format is only understandable by the developers. The business architect does not get a hint, where he has to adjust the models to prevent the error.

The new approach which is contributed by this paper, will find violations without transforming the graph into another format and will be able in the case that a counter example is found to visualize it. Further the validation is repeatable and the rules are formulated in a language similar to BPMN [5], so the persons, who are formulating the specification, which is used for model checking, do only have to learn a few new expressions.

This paper is divided into six sections. The first section above gave a short introduction into the problem domain and the problems which do occur during the certification process for a law or standard. The second section will show how to represent compliance rules in a declarative way and explain the used query language BPMNQ. The third section explains, which steps have to be executed to use the method. Afterwards in section four it is illustrated with a simple example. Section 5 explains the main contribution, which is the automatic derivation of anti pattern. The papers ends with a conclusion in section six. Related work in not discussed in details because the author is not an expert in the general model checking domain.

2 Declarative Representation of Compliance Rules

The next section will explain how a rule is represented in this publication.

Compliance rules are represented by BPMNQ [1]. BPMNQ is a visual query language for business models. In a nutshell BPMNQ is for BPMN, what regular expression are for strings. Like a regular expressions [8] BPMNQ can be used for two purposes:

1. extracting parts of a given business process
2. check if a given process does comply to certain criteria

Like regular expressions are bounded in there expressiveness, the compliance rules, which can be expressed with BPMNQ, are finite too. The focus is on the ordering of the activities in a process, so the rule enforces the process to executed some activities in a certain order. Only the necessary sub set of BPMN and BPMNQ objects are used to achieve this goal

To give this paper a mathematical basis, it will give the most important definitions for the used constructs. It begins which a process graph which can be described as a tuple with some sets for the nodes in the graph and another set with the edges in the graph. The mapping to BPMN should be trivial and therefore is skip here. All node types, which are not mentioned in the definition, are not supported yet.

Definition 1. *A process graph is a tuple $PG = (N, A, E, G, F)$ where*

- *N is a finite set of nodes that is partitioned into the set of activities A, the set of events E, and the set of gateways G. $N = A \cup E \cup G$*
- *The set of events E can be further partitioned into:*
  - *Start events E^s i.e. nodes with no incoming edges.*
  - *Intermediate events E^i*
  - *End events E^e i.e.nodes with no outgoing edges.*
- *$F \subseteq (N \setminus E^e) \times (N \setminus E^s)$ is the sequence flow relation between nodes.*

The query has also one other kind of connections they are called paths. They can be referred as placeholder for activities, which will be later used for creating a match. A match is a sub graph, which is created out of the queried process.

A rule is expressed as a set of BPMNQ queries [1] called pattern. A pair of activities are connected through a path connection and can express an order constraint. This artifact can be run against a repository. This action is shown in Fig. 1.

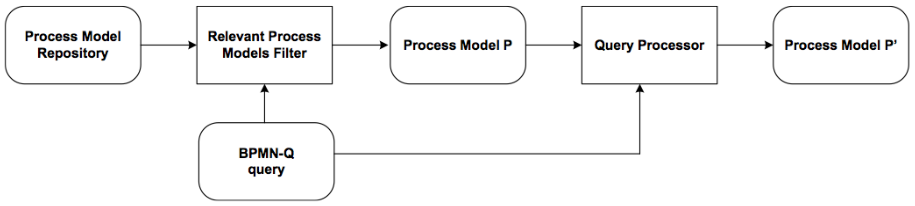

Fig. 1. Processing of a BPMNQ query

Definition 2. *A query graph is a tuple $G = (N, CA, EV, GW, S, NS, P, NP, L, stereotype)$ where*

- *N = finite set of nodes which is partitioned in the sets of concrete activities CA, event nodes EV and gateways GW. $N = CA \cup EV \cup GW$ where*
  - *CA = set of concrete activities.*
  - *EV = set of events.*
  - *GW = set of gateways.*
  - *$CA, EV and GW are disjoint$*
- *$S \subseteq N \times N$ = sequence flow edges between nodes.*

- *$NS \subseteq N \times N$ = negative sequence flow edges between nodes.*
- *$P \subseteq N \times N \times \mathcal{P}(N)$ = path edges between nodes and set of excluded nodes.*
- *$NP \subseteq N \times N$ = negative path edges between nodes.*
- *$L : N \rightarrow l$ is a labeling function.*
- *$stereotype : P \rightarrow \{" << leadsto >> ", " << precedes >> ", \epsilon\}$ assigns semantics to the path*

A process can possibly match to a given query, if the set of the activities in the process graph are a superset of the mentioned activities in the query graph. This property of a model against a query is called, the model is relevant to the query.

Definition 3. *A process graph $PG = (N, A, E, G, F)$ is relevant to query graph $QG = (QN, QCA, QEV, QGW, QS, QNS, QP, QNP, QL, qstereotype) \Leftrightarrow QCA \subseteq A$*

To be able to check a certain ordering in the execution of the process. Some ordered tuples of executions path for every scenario have to be defined. Every execution path begins at a certain start and ends at another end node. Between these activities there are other activities executed.

Definition 4. *An execution path p_{exp} is a sequence of nodes $(n_0, \ldots, n_k)$ where $n_0, \ldots, n_k \in N, n_0 \in E^s$ and $n_k \in E^e$. P_{exp} is a set of all execution paths.*

The numbers 0 to k in an execution path specify the order, in which the activities are executed. To determine the order of the activities a and b in the model, it is necessary to look on all execution paths P_{exp}. It has to be decided for every single path p_{exp} in which order a and b appear. This is done by searching the first occurrence of a from the beginning and also searching the first appearance of b.

Both a and b might appear more then once. This is the case when there are loops or repetitions for other reasons like quality assurance etc. The important thing is that a happens at least once before b.

Definition 5. *An execution ordering relation between nodes on an execution path p_{exp} is defined as $<_{p_{exp}} = \{(n', n'') \in N : n' \in p_{exp}$ [1] $\wedge n'' \in p_{exp} \wedge \exists i, j \in \mathbb{N}\{n' = n_i \wedge n'' = n_j \wedge i < j\} \wedge \nexists j' \in \mathbb{N}\{n'' = n_{j'} \wedge j' < i\}\}$*

The Definition 5 shows how to define ordering. It is still missing how the activities between n' and n'' are received. The next definition will show how to evaluate the BPMNQ path connector which will find all the activities, which are happening between n' and n''. An example is given in Fig. 2.

Definition 6. *A function $subgraph(a, b, p_i) := PSG'(N', E')$, where p_i is a process graph and $a, b \in N_i$, constructs the process sub-graph of p_i where:*

- *$N' = \{x : \forall p_{exp} \in P_{exp}\ (a \in p_{exp} \wedge b \in p_{exp} \wedge x \in p_{exp} \wedge (x = a \vee x = b \vee (a <_{p_{exp}} x \wedge x <_{p_{exp}} b)))\}$*

[1] $n \in p_{exp}$ means that p_{exp} contains n

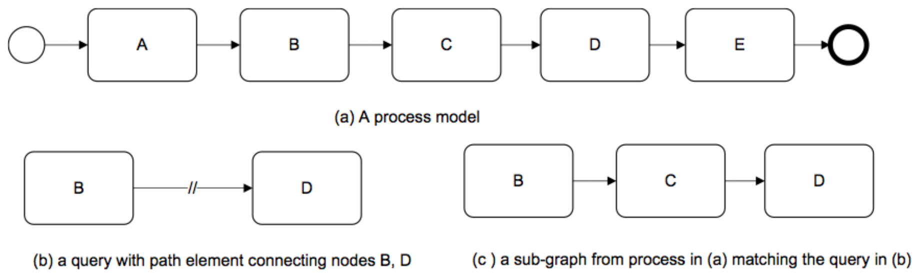
A
B
C
D
E
(a) A process model
B
//
D
(b) a query with path element connecting nodes B, D
B
C
D
(c) a sub-graph from process in (a) matching the query in (b)

Fig. 2. Example evaluation of BPMNQ query

– $\forall x, y \in N' \; (e(x, y) \in F \rightarrow e(x, y) \in E')$

A process graph matches to a given query, if it and the model fulfills the following definition

Definition 7. *A process graph $PG = (N, A, E, G, F)$ matches a query graph QG = (QN, QCA, QEV, QGW, QS, QNS, QP, QNP, QL, qstereotype) iff:*

– $QN \subseteq N$.
– $QS \subseteq F$.
– $\forall (n, m) \in QP \; (subgraph(n, m, PG) \neq \emptyset)$.

These formal definitions are the basis for the following part. An ordering compliance rule contains at least two activities which are connected.

Basic process connections and negative flow sequences are trivial to check so this paper will only give a short explanation how they work. A negative flow rule means that two activities are not allowed to follow in sequence. An example for a negative flow rule would be that on an airlock the sequence ”Open inner door” and ”Open outer door” are never allowed directly after each other (Fig. 3).

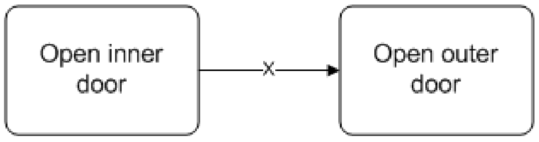

Fig. 3. Example of a negative edge flow

To express now an ordering rule, two activities a and b have to be connected with a path connection. This connection will ensure that both activities are executed in the specified order. An example is shown in Fig 4.

Now the question is if the process in Fig. 5 is considered valid in terms of the given rule (4)? All the time when both activities are executed they are executed

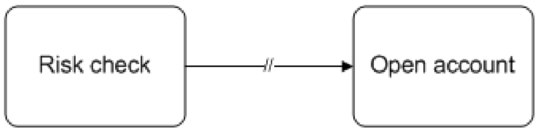

Fig. 4. Example of a simple ordering compliance rule

in the correct order. The compliance rule should be ensure that all the time when an account is opened a risk check was done before. It could happen that this is not the case. If a football player is opening an account, no risk check is made, so the process is not valid, even when all the time the ordering is correct. It is necessary to add more informations to the rule to check it effectively.

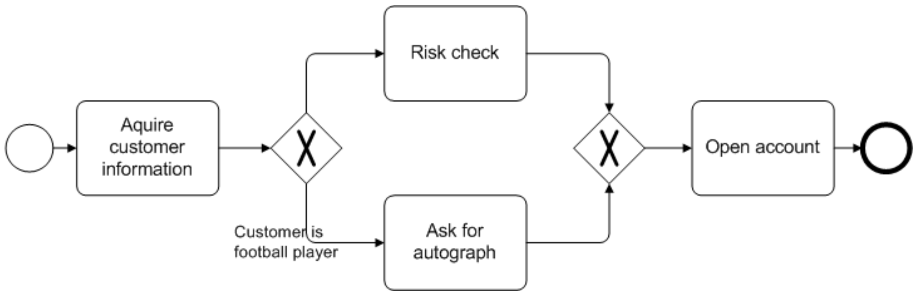

Fig. 5. Example process to open an account from the finance business, which shows miss of information

2.1 Semantics

As it was shown in the paragraph before, it is not sufficient for the validation, that only the plain activities appear in the correct order during the process, further it is important, which gateways do precedes them and which semantics the connection has. Lets assume there is the rule that activity A and activity B have to appear in a process and that both have to be executed in that order, if they are executed. Basically there are two different semantics, which a path between them can have:

1. << *leads to* >> "A leads to B" This means if activity A is executed then B must be executed somewhere afterwards
2. << *precedes* >> "A precedes B" This means whenever B was executed an execution of A has to be in advance

In the example (Fig. 4) it is necessary to add the $<<$ *precedes* $>>$ stereotype to use it as a valid input into the validation process, which is introduced in the next part of the paper. The added annotation is shown in (Fig. 6).

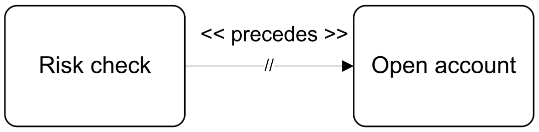

Fig. 6. With $<<$ *precedes* $>>$ extended example from Fig 4

3 Validation Process

In this publication the validation process is executed on one validation rule and one process model as input. These will be used to first give a boolean answer if the model itself is valid according to the rule and in case of a failure the counter example is used to show where the process fails. Further it is also possible to extract relevant process out of a repository using the BPMNQ query.

For a validation rule the following definition is used.

Definition 8. *A validation rule is a pair of an annotated BPMNQ query called pattern and a set of BPMNQ queries called anti-pattern whereat the anti-pattern can be automatically derived from the pattern.*

$$\begin{aligned} V &= (Q_P, A) \\ A &= \{Q_{A_1}, Q_{A_2} \dots Q_{A_i}\} \\ Q_P &\Rightarrow A \end{aligned}$$

3.1 Pattern and Anti Pattern

As said before a pattern is an annotated BPMN query, which has at least two activities. It should ensure that two activities are always executed in a certain order, if they are executed. The pattern will be used to find at least one scenario of the process, where all the activities are done and everything is in the correct order.

If the pattern matches, it cannot be guaranteed that every execution path of the process is valid, because there could still be execution path, which do not comply to the rule. This execution paths are counter examples. To find these counter example it is possible to derive anti pattern from the pattern, which will only match if a violation against the constraint is found. An algorithm to derive these anti pattern is suggested in the Appendix A.

If a process matches the pattern and does not match the anti pattern it is sure that it is valid.

3.2 Step-by-Step Execution

The validation process will give an answer wether the business process complies to certain rules in case of a positive result an "ok" is given otherwise a counter example was found and is used to visualize the violation. The steps are done in the following order:

1. identify process, which should be used for validation process
2. create pattern
3. run pattern against process to ensure that at least one valid execution path exists
4. if no match is found, validation failed
5. derive anti pattern
6. run anti pattern against process to find counter example
7. if one is found, visualize violation with match otherwise accept

4 Generation of Anti Pattern

The generation of the anti pattern is the big contribution of this paper. It is based on an inductive principle, that there is a set of basic patterns, where the anti pattern is known, and then in a structural step, where the process model is divided into these basic patterns. The anti patterns are collected in a list and at the end of the execution all anti patterns are evaluated against the model to find any counter example. The know basic pattern are shown in Fig. 7, further they can also be found as encoded process graph in the appended algorithm (Appendix A)

5 Example Execution and visualization of Counter Examples

The next section will run through an example and apply the validation process.

The model is taken from an order fulfillment scenario in the retailer business, which is shown in Fig. 8. It receives an order and then follows two parallel execution paths, to bill and pack the order, before it is send to the customer. The process does not comply to the validation rule, which is proposed in the next paragraph. The problem is, that it could happen that an only partly packed package is send to the customer.

Step 1: Identify process

The process, which was used in the paragraph before, will be used for validation. The process is already finally created, it would also be possible to use the validation already during the designing process in realtime. When the order or the amount of the activities, are changed the whole validation can be run again.

Step 2: create pattern

The merchant decides in his strategy meeting that they only want to send out

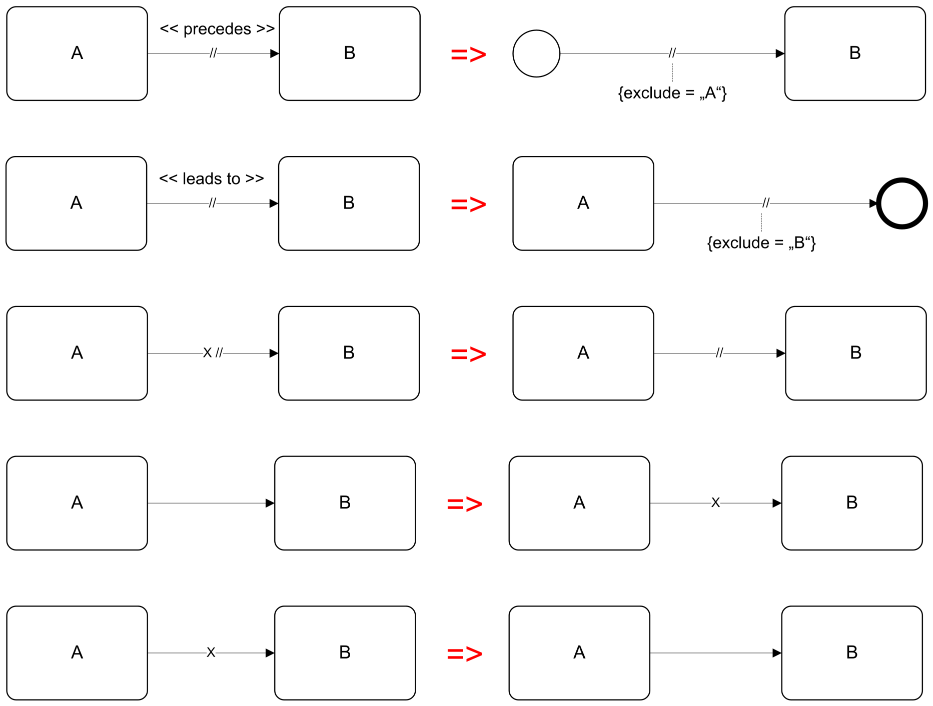

Fig. 7. Basic rules for deriving anti pattern

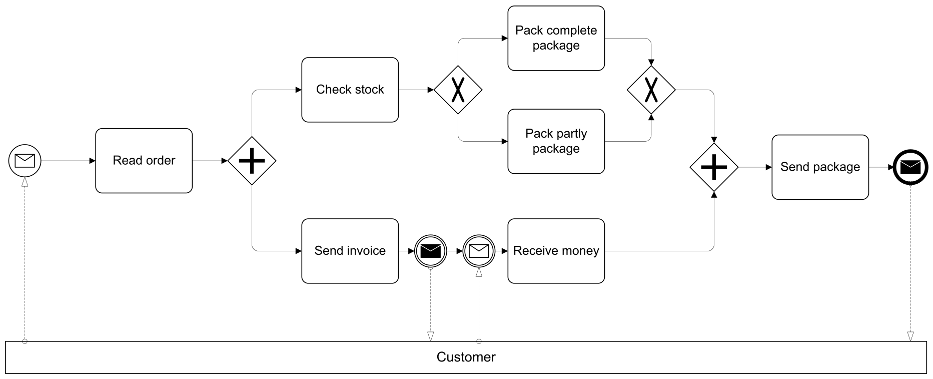

Fig. 8. Example of an fulfillment process of a retailer

complete packages and that every package has to be billed and payed in advance. The generated pattern by this sentence is shown in Fig. 9. The pattern contains the 4 activities ”Send invoice”, ”Receive money”, ”Pack complete package” and ”Send package”. All 4 activities has to happen in a process. The ”Send invoice” activity must be followed somewhere by the ”Receive money” activity because this is only a << *leadsto* >> path the ”Receive money” can also happen independently from the ”Send invoice”. When the ”Send package” activity is executed ”Pack complete package” and ”Receive money” have to be executed in advance.

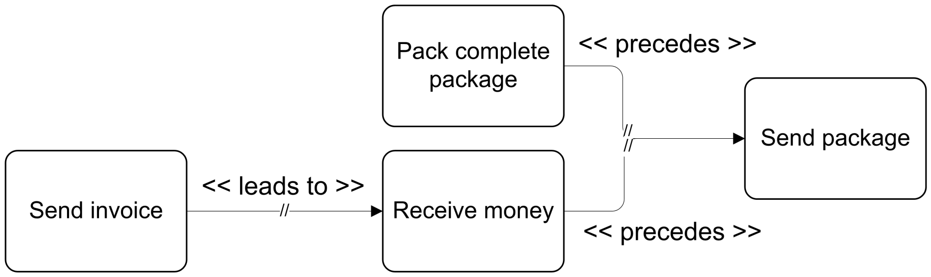

Fig. 9. Example compliance rules for fulfillment process

Step 3: run pattern against process, to ensure that at least one valid execution path exists
The pattern is executed and one valid execution path (Fig. 10) is found. The ”Receive money” and ”Pack complete package” activity are executed in parallel but this is a valid execution scenario according to the rule. The match ends with the ”Send package”, which is preceded by the two required activities. The ”Send invoice” does also lead to the specified activities.

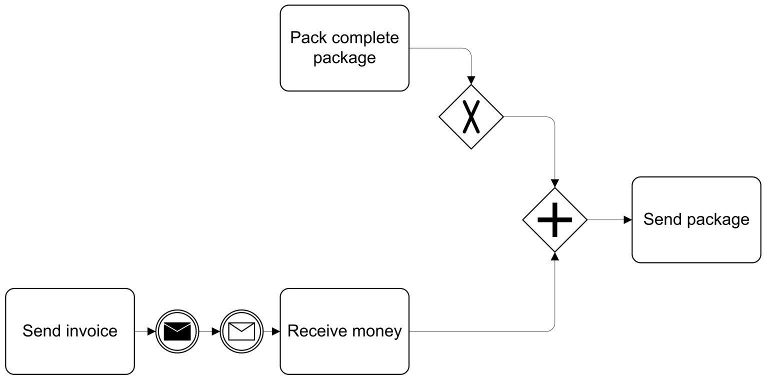

Fig. 10. Example match of the pattern

Step 4: if no match is found, validation failed
In this case it was possible to find a match (Fig. 10). This means that at least one possible scenario complies to the proposed rule. The anti pattern have to be generated and executed, to proof that no counter examples can be found. The validation process can continue.

Step 5: derive anti pattern
The algorithm (Appendix A) is used to derive all anti pattern. Three anti pattern (Fig. 11) are found for the validation rule. There are three path connections, these will be used during the derivation process. Every path will create its own anti pattern. Every anti pattern check for a certain constraint. In this example the anti pattern try to find the following cases:

1. Find all path from "Send invoice" to the end which do not contain a "Receive money" activity
2. Find all path from the start to the activity "Send package" which do not contain the "Receive money" activity
3. Find all path from the start to the activity "Send package" which do not contain the "Pack complete package" activity

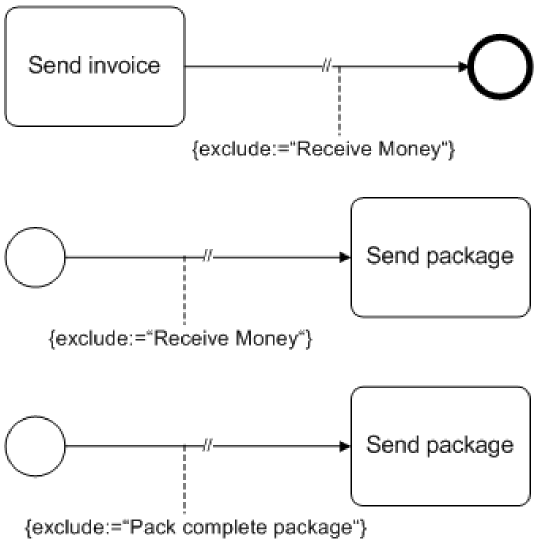
Send invoice
{exclude:="Receive Money"}
Send package
{exclude:="Receive Money"}
Send package
{exclude:="Pack complete package"}

Fig. 11. Example anti pattern

Step 6: run anti pattern against process to find counter example
In the process a counter example can be found. When a product is not enough on stock nevertheless it is send out even if it is only partly packed. The third rule finds a match. This match can now be used to show the violation.

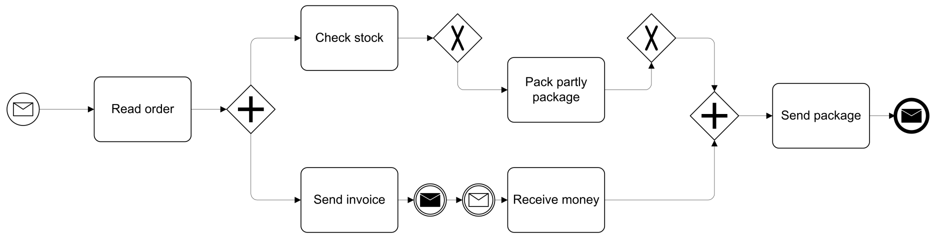

Fig. 12. Counter example for validation rules

Step 7: if one is found visualize violation with match otherwise accept
The match which was found in Step 6 (Fig. 12) is used to give the business architect a hint, where he has to adjust the model to comply to the rule. To do so the counter example is taken, the lines are drawn thicker and to have a warning contrast the activities are colored in red (Fig. 13).

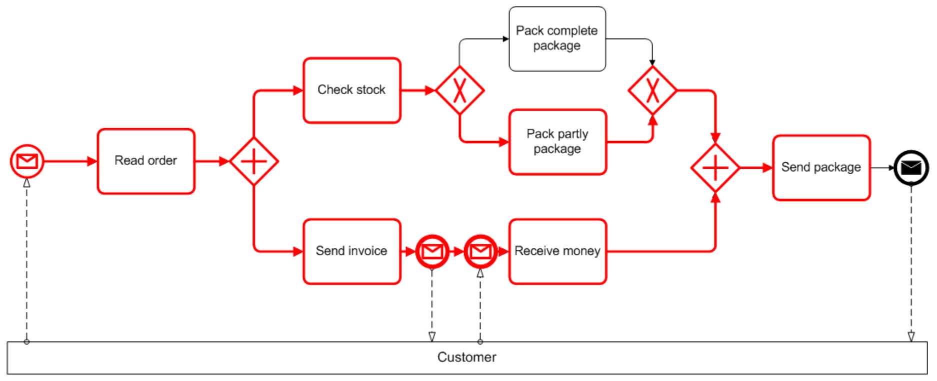

Fig. 13. Process with visualized counter example

After finding this violation the business architect has to work again on the process and repeat the validation.

6 Conclusion

A cheap way of validating an process against certain ordering compliance rules was shown. During the validation process counter examples are searched and later used for visualization of constraint validation. The visualization is a big help for humans, who are able to understand, which problems were found during checking, and they are able to adjust the process, to later fit into the constraints. Further it is not necessary for them to learn a new methodology. There is only the new path constraint. The model checking is language independent and it would be even possible to use it for other graph based workflow languages. It is possible to implement the approach into model repository, which validate all the process or just use it for a single process to visualize the constraint violation. A prototypical implementation was done, which shows that the approach works.

A Algorithm

Listing 1.1. Formalized algorithm to generate anti-pattern

```
A = [] // Initialiaze list of anti-pattern
for path in P: // iterate through every path "-//->"
    if(stereotype(path) == "<<␣precede␣to␣>>"):
        A.add(
            // N finite set of nodes
            {Start, path.end},
            // S sequence flow edges between nodes
            {},
            // NS negative sequence flow edges between nodes
            {},
            // P path edges between nodes, including excluded Nodes
            {(Start, path.end, {path.start})},
            // NP negative path edges between nodes
            {},
            // l is a labeling partial function
            {(path.end, path.l(path.end))},
            // no stereotypes anymore
            {(..., "")}
        )
    else if(stereotype(path) == "<<␣leads␣to␣>>"):
        A.add(
            // N finite set of nodes
            {path.start, End},
            // S sequence flow edges between nodes
            {},
            // NS negative sequence flow edges between nodes
            {},
            // P path edges between nodes, including excluded Nodes
            {(path.start, End, {path.end})},
            // NP negative path edges between nodes
            {},
            // l is a labeling partial function
            {(path.start, path.l(path.start))},
            // no stereotypes anymore
            {(..., "")}
        )
for negPath in NP: // iterate through every negative path "-X//->"
    A.add(
        // N finite set of nodes
        {negPath.start, negPath.end},
        // S sequence flow edges between nodes
        {},
        // NS negative sequence flow edges between nodes
        {},
        // P path edges between nodes, including excluded Nodes
        {(negPath.start, negPath.end, {})},
        // NP negative path edges between nodes
        {},
        // l is a labeling partial function
        {(negPath.start, negPath.l(negPath.start)),
            (negPath.end, negPath.l(negPath.end)},
        // no stereotypes anymore
        {(..., "")}
    )
for flow in S: // iterate through every sequence flow "-->"
    A.add(
        // N finite set of nodes
        {flow.start, flow.end},
        // S sequence flow edges between nodes
        {},
        // NS negative sequence flow edges between nodes
        {(flow.start, flow.end)},
        // path edges between nodes, including excluded Nodes
        {},
        // negative flow edges between nodes
        {},
        // l is a labeling partial function
        {(flow.start, flow.l(flow.start)), (flow.end, flow.l(flow.end)},
        // l is a labeling partial function
        {(flow.start, path.l(flow.start))},
        // no stereotypes anymore
        {(..., "")}
    )
for negFlow in S: // iterate through every negative sequence flow "-X->"
    A.add(
        // N finite set of nodes
        {negFlow.start, negFlow.end},
        // S sequence negFlow edges between nodes
        {(negFlow.start, negFlow.end)},
        // NS negative sequence negFlow edges between nodes
        {},
        // path edges between nodes, including excluded Nodes
        {},
        // negative negFlow edges between nodes
        {},
        // T assignment of types to the nodes)
        {(negFlow.start, "CONCRETE_ACTIVITY"),
            (negFlow.end,"CONCRETE_ACTIVITY")},
        // l is a labeling partial function
        {(negFlow.start, negFlow.l(negFlow.start)),
            (negFlow.end, negFlow.l(negFlow.end)},
         // l is a labeling partial function
        {(negFlow.start, path.l(negFlow.start))},
        // no stereotypes anymore
        {(..., "")}
    )
```

References

[1] *Ahmed Awad: BPMN-Q: A Language to Query Business Processes. EMISA 2007: 115-128*

[2] *Ahmed Awad, Gero Decker, and Mathias Weske: Efficient Compliance Checking using BPMN-Q and Temporal Logic. 6th International Conference on Business Process Management BPM 2008*

[3] *Aditya Ghose and George Koliadis: Auditing Business Process Compliance*

[4] *Rik Eshuis: Symbolic Model Checking of UML Activity Diagrams* Shazia Sadiq, Guido Governatori, Kioumars Naimiri: Modeling Control Objectives for Business Process Compliance

[5] *OMG BPMN 1.1 - OMG Final Adopted Specification, January 2008*

[6] *Aufschwungserwartungen: IT-Gehälter 2008, Heise iX 6/2008*

[7] *Nicht besetzbare Stellen für beruflich Hochqualifizierte in Deutschland – Ausmaß und Wertschöpfungsverluste, Oliver Koppel, IW-Trends 1/2008*

[8] *Larry Wall. Apocalypse 5: Pattern Matching. http://dev.perl.org/perl6/doc/design/apo/A05.html 2002-06-04*