

**TRƯỜNG ĐẠI HỌC CÔNG NGHỆ**

**KHOA CÔNG NGHỆ THÔNG TIN**



**CÔNG TRÌNH THAM DỰ  
HỘI NGHỊ SINH VIÊN NGHIÊN CỨU KHOA HỌC  
CẤP KHOA  
NĂM HỌC 2016 – 2017**

Tên công trình: A method to validate business process models

Người hướng dẫn: TS. Đặng Đức Hạnh

**Nhóm thực hiện**

1. Nguyễn Thị Lương – K58CLC
2. Phạm Nguyễn Hoàng – K59CA
3. Lương Ngọc Huyền – K59CA

## Table of contents

1.	Introduction .....	- 3 -
2.	Motivating example and background .....	- 4 -
3.	Approach overview .....	- 5 -
4.	Main techniques .....	- 6 -
4.1.	A first catalogue of patterns .....	- 6 -
4.1.1.	Deadlock pattern.....	- 6 -
4.1.2.	Multiple termination pattern.....	- 8 -
4.1.3.	Implicit gateway pattern.....	- 8 -
4.1.4.	Combined gateway pattern.....	- 9 -
4.1.5.	Loop pattern .....	- 10 -
4.2.	Test case generation .....	- 10 -
5.	Tool support and discussion .....	- 12 -
5.1.	Implementation.....	- 12 -
5.1.1.	BPMN metamodel.....	- 13 -
5.1.2.	BPMN Parser.....	- 14 -
5.1.3.	Deadlocks and multiple terminations .....	- 14 -
5.1.4.	Anti-patterns .....	- 15 -
5.2.	Discussion .....	- 16 -
6.	Related work.....	- 17 -
7.	Conclusion and future work .....	- 18 -
	References .....	- 19 -

## Table of figures

Figure 1. A business process model containing a deadlock (left) and a multiple termination (right). These errors are ignored by a large number of model designers as they lack the ability to validate models. ....	- 4 -
Figure 2. XML representation of a business process model (a) and the same model shown in USE's object diagram (b). A deadlock is detected at parallelgateway1 (highlighted in red).....	- 6 -
Figure 3. Visualisation of the deadlock pattern.....	- 7 -
Figure 4. Visualisation of the multiple termination pattern. ....	- 8 -
Figure 5. Visualisation of the implicit gateway pattern. In this example, Service Task and Script Task are being used as gateways.....	- 9 -
Figure 6. A business process model containing a combined parallel gateway (second from left). ....	- 9 -
Figure 7. A business process model containing a loop. ....	- 10 -
Figure 8. Result after running the <code>attemptLoop()</code> process. All snapshots failed to satisfy the <code>NoLoops</code> constraint. ....	- 12 -
Figure 9. BPMNChecker's actions represented as menu items (left) and toolbar icons (right) in USE.....	- 13 -
Figure 10. Our implementation of the BPMN specification, visualised as a class diagram in USE. ....	- 13 -
Figure 11. BPMN Parser's file selection dialogue.....	- 14 -
Figure 12. Detection of deadlocks and multiple terminations using USE. Results are shown in a dialogue box (a). The models in figure 4.1. and 4.2. are visualised as object diagrams in (b) and (c), respectively.....	- 15 -
Figure 13. Visualisations of business process models in 4.5 (figure a), 4.4 (figure b) and 4.3 (figure c) as USE object diagrams. OCL query results are shown in red. ....	- 16 -
Figure 14. A business process model containing no start event (a) and its corresponding object diagram (b).....	- 17 -

## 1. Introduction

The modelling of business processes is becoming increasingly popular. Business process models specify how a business works with understandable and simple notations. Business Process Modelling is the set of activities conducted for visually depicting qualitatively grounded models of organisation's processes, so that processes can be analysed, monitored and improved regarding their expected value [1]. Process modelling is concerned with the representation of processes that take place within and between organizations, by means of process models.

Nowadays, these models are widely used and play an important role in business and organizations. The importance of business process modelling quality motivated several researchers to propose solutions to improve the effectiveness and qualification of business process models [2]. Ensuring compliance of business processes with regulations, policies and quality constraints is a challenging task. A business model should satisfy the following qualities: correctness, integrity, modifiability, complexity and understandability [2].

With the important role and efficiency of the business process model, scientists, organisations and companies have researched and produced many standards and frameworks to evaluate the quality of a business process model (BPM). Software tools are used to define, manage, ensure and improve the quality of business models.

The Object Management Group (OMG) has developed a standard Business Process Model and Notation (BPMN) to standardize a business process model and notation in the face of many different modelling notations and viewpoints. BPMN provides a notation that is readily understandable by all business users, from the business analysts that create the initial drafts of the processes, to the technical developers responsible for implementing the technology that will perform those processes. BPMN create a standardized bridge for the gap between the business process design and process implementation.

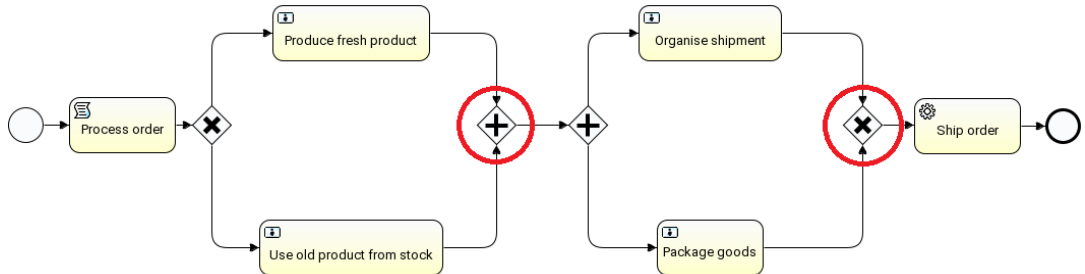
However, the utilities to validate BPMN models are limited in number and quality. Therefore, we have proposed a new approach to solving the

business process model validation problem, especially detecting deadlocks and anti-patterns.

A deadlock in business process model exists when a certain instance of the model is not able to continue working and complete the process. The proposed approach is based on queries for BPMN models using OCL (Object Constraints Language). OCL is a declarative and predicate logic like language that supplements the UML. We present a set of so-called deadlock patterns whose occurrence in process models usually leads to deadlocks and an algorithm for detecting these patterns.

The paper is structured as follows. We first extend the motivating example in section 2. The contribution is presented in section 3. In the section 4, we discuss and explain the main technique used. Section 5 gives information on tool support and discussion. Related work is present in section 6. Finally, the paper is concluded with a discussion of future work in section 7.

## 2. Motivating example and background



*Figure 1. A business process model containing a deadlock (left) and a multiple termination (right). These errors are ignored by a large number of model designers as they lack the ability to validate models.*

Figure 1 shows an example of a business process model. The aim of business process modelling is to improve operational and manage business processes. Due to their visual nature, business process models have gained popularity with among business managers and analysts alike. BPMN provides businesses with the capability of understanding their internal business procedures in a graphical notation and gives organizations the ability to communicate these procedures in a standard manner [1].

Due to the important role of business process modelling, the validation of process models becomes ever more important. At present, a large number of

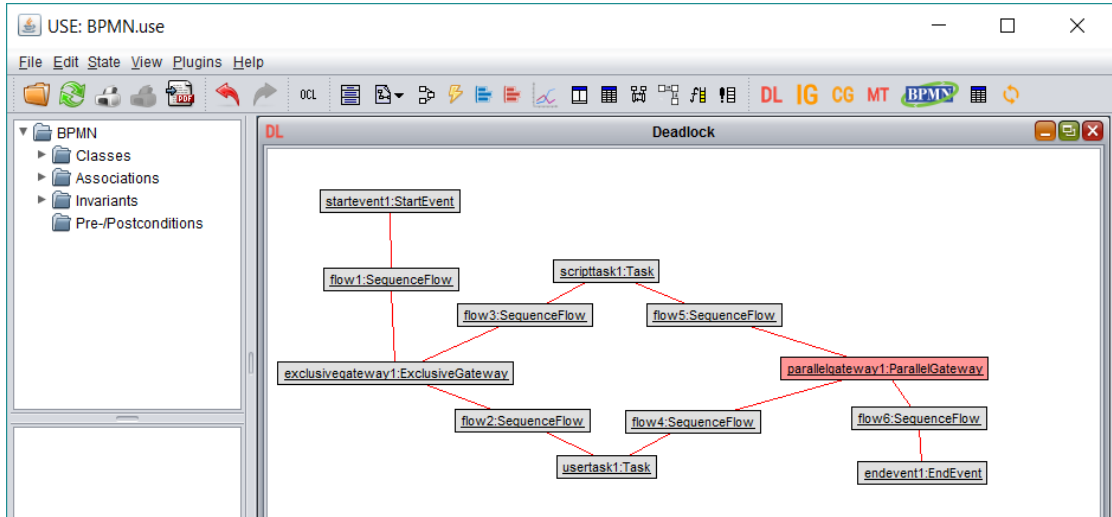
tools handle the creation and visualisation of business process models using BPMN, but most of them cannot detect deadlocks in BPMs and evaluate the effectiveness of business processes through models. The errors in the above example cannot be detected by most process model designers. For more complex models, these errors are impossible to detect manually and therefore the models cannot be put into practical use. The purpose of our approach is to deal with this problem.

### 3. Approach overview

In order to detect deadlocks as well as other anti-patterns in business process models, we utilise USE – a tool for validating object-oriented systems. Our approach can be described as follows. Firstly, business process models are created under the XML file format, which can be done by tools such as Activiti Designer by Alfresco, an Eclipse plug-in. Secondly, the XML file is loaded into USE as a model. This requires the use of a custom plug-in that we have developed. Thirdly, the model is evaluated using OCL queries, resulting in a list of elements matching the proposed patterns. Finally, the model is visualised in an object diagram, with error-triggering elements highlighted, which serves as a guide for process modellers to rectify errors.

```
<process id="myProcess" name="My process" isExecutable="true">
  <startEvent id="startevent1" name="Start"/>
  <endEvent id="endevent1" name="End"/>
  <exclusiveGateway id="exclusivegateway1" name="Exclusive Gateway"/>
  <sequenceFlow id="flow1" sourceRef="startevent1" targetRef="exclusivegateway1"/>
  <parallelGateway id="parallelgateway1" name="Parallel Gateway"/>
  <userTask id="usertask1" name="User Task"/>
  <scriptTask id="scripttask1" name="Script Task" activiti:autoStoreVariables="false"/>
  <sequenceFlow id="flow2" sourceRef="exclusivegateway1" targetRef="usertask1"/>
  <sequenceFlow id="flow3" sourceRef="exclusivegateway1" targetRef="scripttask1"/>
  <sequenceFlow id="flow4" sourceRef="usertask1" targetRef="parallelgateway1"/>
  <sequenceFlow id="flow5" sourceRef="scripttask1" targetRef="parallelgateway1"/>
  <sequenceFlow id="flow6" sourceRef="parallelgateway1" targetRef="endevent1"/>
</process>
```

a



b

Figure 2. XML representation of a business process model (a) and the same model shown in USE's object diagram (b). A deadlock is detected at parallelgateway1 (highlighted in red).

## 4. Main techniques

This section introduces our main techniques in detecting deadlocks and anti-patterns in business process models.

### 4.1. A first catalogue of patterns

In this section, we present a set of (anti-)patterns in BPMN, and how to detect these patterns using OCL queries.

#### 4.1.1. Deadlock pattern

We define the optionality of a flow or flow node as follows: If a flow/node is reachable from an exclusive/inclusive gateway with an even number of matching split/join gateways in between, then it can be executed or not, which means it is optional. Otherwise, a flow/node must always be executed and thus not optional.

A deadlock happens when an optional flow goes into a parallel gateway, which requires that all flows going into it be executed. This is often caused by unbalanced gateway pairs, e.g. when an exclusive gateway splits a process into multiple paths and a parallel gateway merges them, as in Figure 3 below. In such cases, the execution cannot proceed.

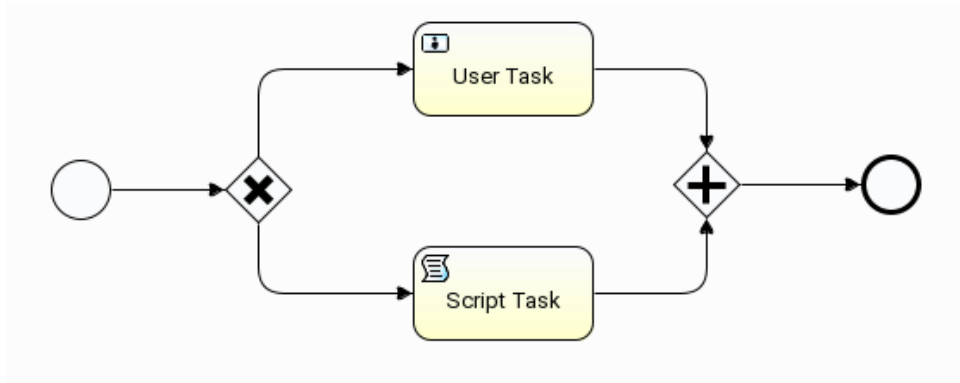


Figure 3. Visualisation of the deadlock pattern.

Detecting deadlocks and multiple terminations in business process models requires more than simple OCL queries. Therefore, we have utilised helper methods and attributes:

**isVisited** : Boolean shows which node has been visited. It is used to prevent infinite recursion in models with loops.

**optional** : Sequence(Boolean) shows the optionality of a flow node or sequence flow. After a split or fork, **optional** is appended with **true** or **false** depending on the gateway type (**true** for exclusive/inclusive gateways, **false** otherwise). After a join or merge, the last value is removed from the sequence. A start event's **optional** value is always Sequence{false}.

**label()** is an operation that changes sequence flows' **optional** value. It works as follow:

- If there is no split/fork from a sequence flow's source node, its **optional** is equal to that of the source node. Otherwise, the sequence flow's **optional** value is appended with **true** or **false** according to the gateway type.
- If there is no merge or join from the sequence flow to its target node, the target node's **optional** value is equal to that of the sequence flow. Otherwise, the last boolean value of **optional** is removed, signifying a decrease in depth.
- The operation is applied recursively on the next sequence flows coming from unvisited flow nodes. This prevents the algorithm from repeating infinitely when it encounters a loop.



After applying `label()`, each sequence flow and flow node's `optional` reflects its optionality. If a node or flow is optional, the last element of `optional` is `true`; otherwise, it is `false`. If there is a loop or multiple sources, `optional` is an empty Sequence. An OCL query can now be executed to detect deadlocks:

```
ParallelGateway.allInstances()->select(a |  
a.incoming->size() > 1 and a.incoming->exists(flow |  
flow.optional->at(flow.optional->size()))
```

#### 4.1.2. Multiple termination pattern

Multiple terminations are the opposite of deadlocks - they happen when a required flow goes into an exclusive or inclusive gateway [3]. Figure 4 is an example of multiple termination:

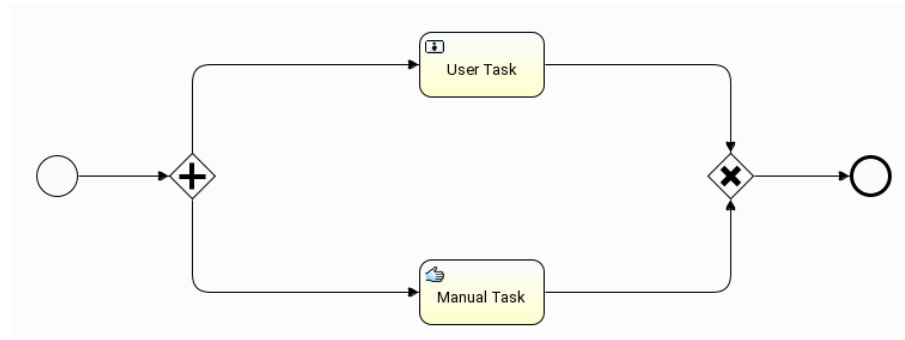


Figure 4. Visualisation of the multiple termination pattern.

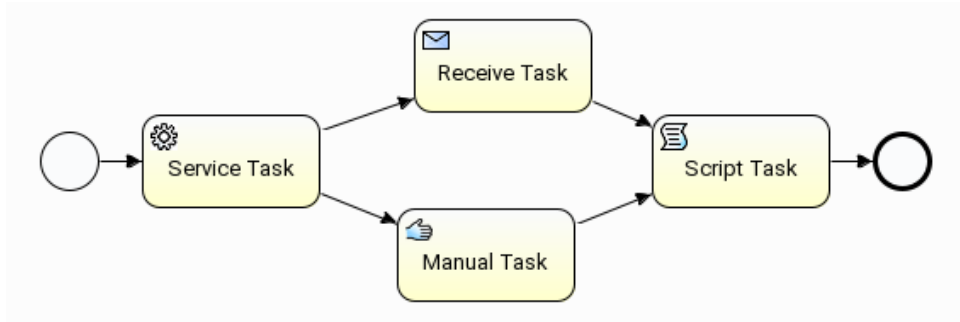
Our technique for detecting multiple terminations is similar to the aforementioned deadlock detection technique. Sequence flows and flow nodes are labelled with their optionality value. OCL queries are carried out to acquire the list of gateways where multiple terminations occur:

```
ExclusiveGateway.allInstances()->select(a | a.incoming-  
>size() > 1 and a.incoming->exists(flow | flow.optional-  
>at(flow.optional->size()) = false))  
InclusiveGateway.allInstances()->select(a | a.incoming-  
>size() > 1 and a.incoming->exists(flow | flow.optional-  
>at(flow.optional->size()) = false))
```

#### 4.1.3. Implicit gateway pattern

Implicit gateways are characterised by events or tasks having more than one incoming/outgoing flow. Despite being valid parts of a business process

model, implicit gateways are considered anti-patterns and their usage is discouraged because they create ambiguity.



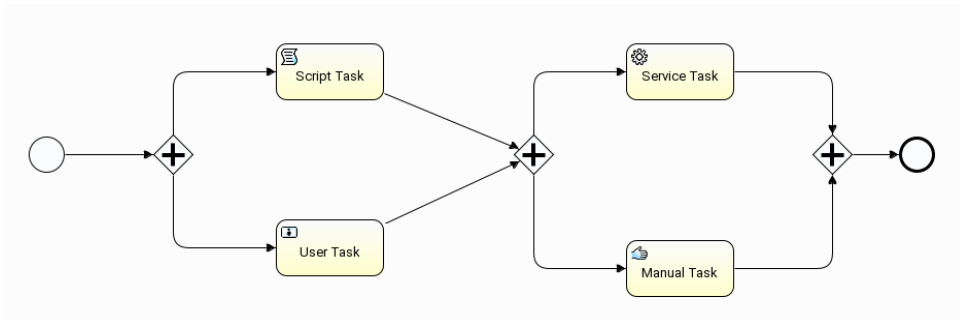
*Figure 5. Visualisation of the implicit gateway pattern. In this example, Service Task and Script Task are being used as gateways.*

Tasks and events used as implicit gateways can be found by executing these OCL queries:

```
Task.allInstances()->select(a | a.incoming->size() > 1 or  
a.outgoing->size() > 1)  
Event.allInstances()->select(a | a.incoming->size() > 1 or  
a.outgoing->size() > 1)
```

#### 4.1.4. Combined gateway pattern

Combined gateways are gateways with more than one flow on both sides (incoming flows and outgoing flows).



*Figure 6. A business process model containing a combined parallel gateway (second from left).*

The following OCL query can be executed to detect the presence of combined gateways in the model:

```
Gateway.allInstances()->select(a | a.incoming->size() > 1  
and a.outgoing->size() > 1)
```

#### 4.1.5. Loop pattern

While loops are often necessary to accurately model business processes, the process models are often clearer if the loops are encapsulated in a sub-process.

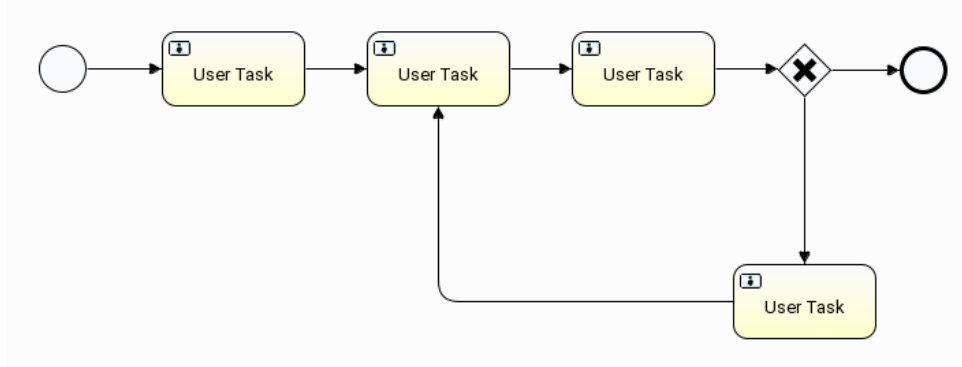


Figure 7. A business process model containing a loop.

If two nodes are reachable from each other, there are loops in the model. We use OCL's `closure()` to recursively select reachable nodes:

```
FlowNode.allInstances()->select(a | Sequence{a}->closure(outgoing.targetRef)->exists(b | b<>a and Sequence{b}->closure(outgoing.targetRef)->includes(a)))
```

#### 4.2. Test case generation

In USE, ASSL (A Snapshot Sequence Language) is used to automatically generate and validate system snapshots [4]. In this paper, we give an example of test case generation for the aforementioned loop detection technique.

Firstly, as snapshots are checked against invariants, the OCL query in Section 4.1.5 needs to be rewritten as a metamodel invariant:

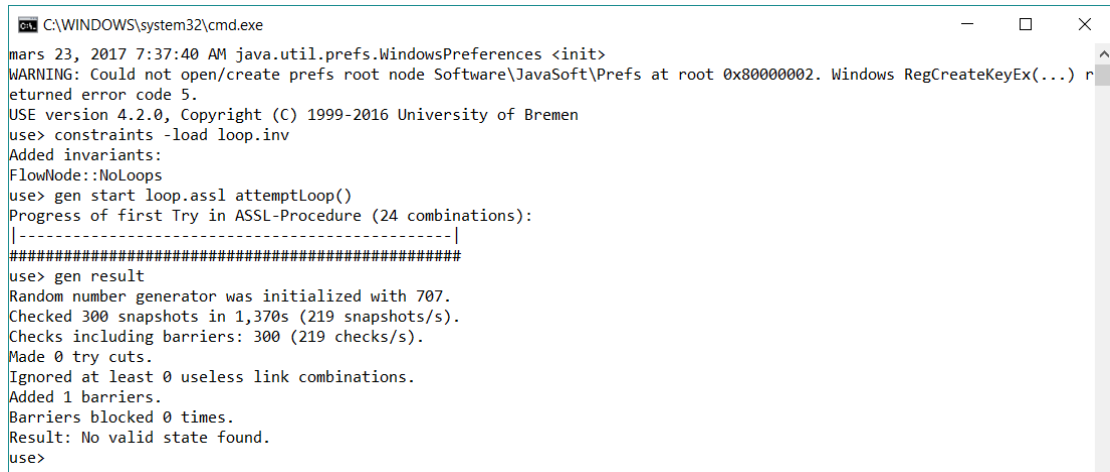
```
context FlowNode
  inv NoLoops:
    Sequence{self}.closure(outgoing.targetRef)->exists(b | b<>self and Sequence{b}.closure(outgoing.targetRef)->includes(self)) = false
```

In order to test this invariant, we have written an ASSL procedure that creates BPMN processes containing a loop with different sizes:

```
procedure attemptLoop()
```

```
var s: StartEvent, e: EndEvent, ts: Sequence(Task),
f: SequenceFlow, sf: SequenceFlow, t1: Task, t2: Task,
x: Integer , y: Integer;
begin
  s:=Create(StartEvent);
  e:=Create(EndEvent);
  ts:=CreateN(Task, [25]);
  for j: Integer in [Sequence{1..24}]
  begin
    f := Create(SequenceFlow);
    Insert(OutgoingFlow, [ts->at(j)] , [f]);
    Insert(IncomingFlow, [ts->at(j+1)], [f]);
  end;
  f := Create(SequenceFlow);
  Insert(OutgoingFlow, [s] , [f]);
  Insert(IncomingFlow, [ts->at(1)], [f]);
  f := Create(SequenceFlow);
  Insert(OutgoingFlow, [ts->at(25)], [f]);
  Insert(IncomingFlow, [e] , [f]);
  x := Try([Sequence{2..25}]);
  y := Try([Sequence{1..x-1}]);
  t1 := [ts->at(x)];
  t2 := [ts->at(y)];
  sf := Create(SequenceFlow);
  Insert(OutgoingFlow, [t1], [sf]);
  Insert(IncomingFlow, [t2], [sf]);
end;
```

The above procedure creates a process with a series of 25 tasks, and then tries to make a loop by adding a sequence flow from a task to another task executed before it. As a result,  $25C2 = 300$  snapshots were created and checked. If USE finds a valid system state, the test is terminated. The result is shown by executing `gen result`. In figure 8 below, USE found no valid state, which means that the test is passed.



```
C:\WINDOWS\system32\cmd.exe
mars 23, 2017 7:37:40 AM java.util.prefs.WindowsPreferences <init>
WARNING: Could not open/create prefs root node Software\JavaSoft\Prefs at root 0x80000002. Windows RegCreateKeyEx(...) r
returned error code 5.
USE version 4.2.0, Copyright (C) 1999-2016 University of Bremen
use> constraints -load loop.inv
Added invariants:
FlowNode::NoLoops
use> gen start loop.assl attemptLoop()
Progress of first Try in ASSL-Procedure (24 combinations):
|-----|
#####
use> gen result
Random number generator was initialized with 707.
Checked 300 snapshots in 1,370s (219 snapshots/s).
Checks including barriers: 300 (219 checks/s).
Made 0 try cuts.
Ignored at least 0 useless link combinations.
Added 1 barriers.
Barriers blocked 0 times.
Result: No valid state found.
use>
```

*Figure 8. Result after running the `attemptLoop()` process. All snapshots failed to satisfy the `NoLoops` constraint.*

## 5. Tool support and discussion

In this section we present our implementation of a business process model validation tool – BPMNChecker. It is available as a plug-in for USE and utilises USE's OCL and SOIL capabilities.

### 5.1. Implementation

USE is an environment for the specification of information systems. It is based on a subset of UML. A USE specification consists of a textual description of a model conforming to UML; additional constraints, including pre- and post-conditions and invariants. In later versions, specifications can include operation implementations.

The Object Constraint Language (OCL) is a declarative language describing constraints on UML models. OCL is employed in USE to specify model invariants and pre-/post-conditions, as well as executing queries on systems. In BPMNChecker, OCL expressions are used to check for elements that can trigger errors or anti-pattern warnings.

SOIL (Simple OCL-like Imperative Language) [5] is the imperative programming language of USE. It can be used to define operations imperatively and change the system state of a USE specification.

USE has APIs for the definition of plug-in actions, which are Java classes implementing the `IPlugin` interface. They can be displayed as

buttons or menu items inside the program's main window. In addition, USE provides APIs for changing the system state and validating OCL expressions from inside the plugin. We have taken advantage of this functionality to create checkers for deadlocks and anti-patterns in BPM. The results are displayed using an extended version of USE's object diagram that highlights elements with deadlocks or anti-patterns.

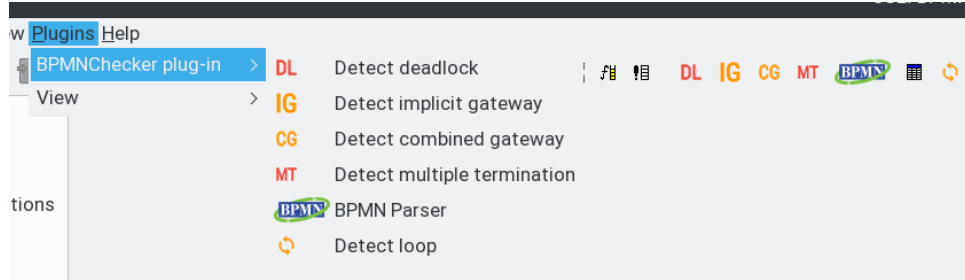


Figure 9. BPMNChecker's actions represented as menu items (left) and toolbar icons (right) in USE.

### 5.1.1. BPMN metamodel

To validate BPMs using USE, a metamodel specification must be defined. We have chosen to implement a subset of the BPMN specification, keeping only the relevant elements and attributes.

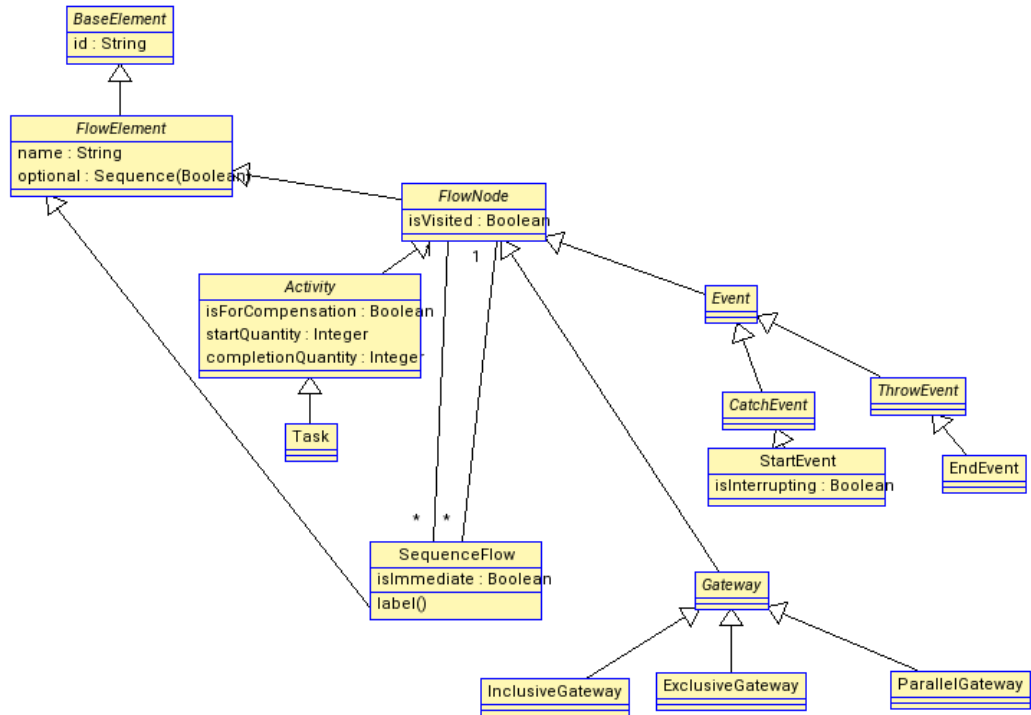


Figure 10. Our implementation of the BPMN specification, visualised as a class diagram in USE.

### 5.1.2. BPMN Parser

A business process model can be represented as an XML file. As of March 2017, the latest version of BPMN is 2.0.2, whose schema is defined by the Object Management Group [6]. In order to retrieve information from BPMN XML files, Camunda BPMN model API [7] is used.

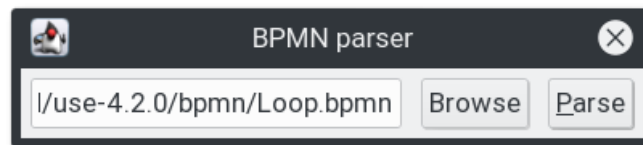


Figure 11. BPMN Parser's file selection dialogue.

### 5.1.3. Deadlocks and multiple terminations

In order to detect deadlocks and multiple terminations, `label()` is applied to sequence flows coming out of start events:

```
! for se in StartEvent.allInstances do for fl in  
se.outgoing do fl.label() end end
```

Before running `label()` recursively, the system's state is initialised as follows:

```
! for fe in FlowElement.allInstances do fe.optional :=  
Sequence{} end  
! for node in FlowNode.allInstances do node.isVisited :=  
false end  
! for se in StartEvent.allInstances do se.optional :=  
Sequence{false} end
```

The commands are run inside the shell made available to plug-ins. After running one of the plug-in's actions, a dialog is shown indicating the number of errors detected. If there are errors, the user can choose to visualise them using an object diagram.

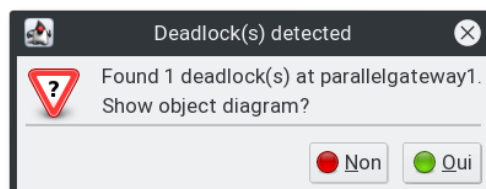
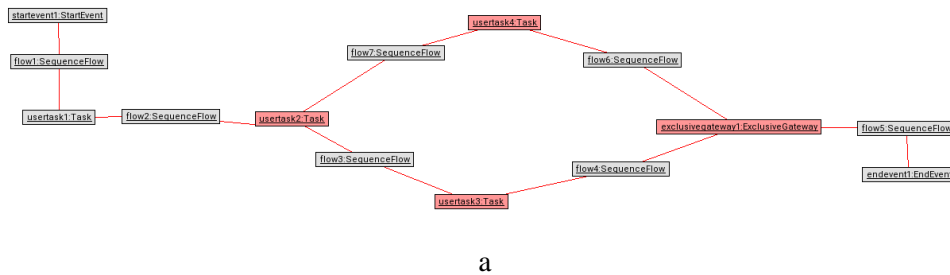




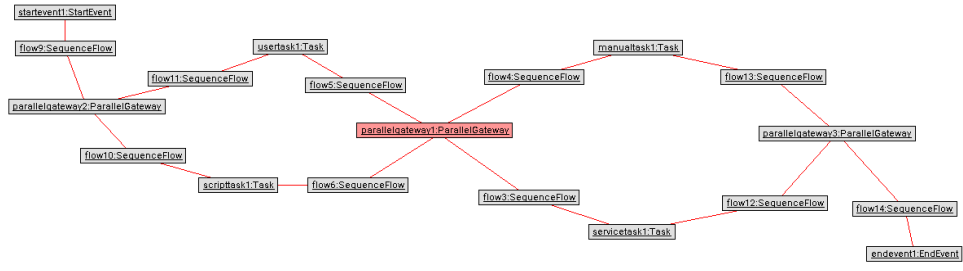
Figure 12. Detection of deadlocks and multiple terminations using USE. Results are shown in a dialogue box (a). The models in figure 4.1. and 4.2. are visualised as object diagrams in (b) and (c), respectively.

#### 5.1.4. Anti-patterns

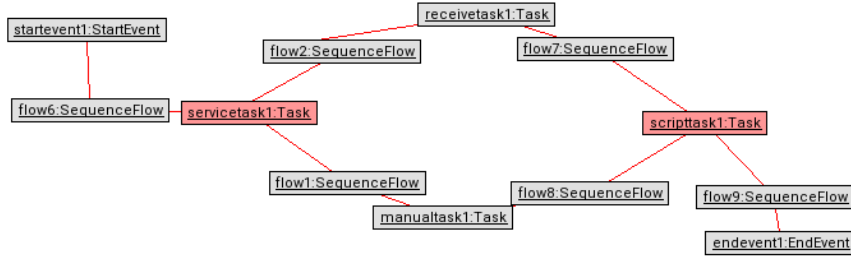
Anti-patterns such as loops, implicit gateways and combined gateways can be detected by OCL queries. They can be entered from USE's OCL expression evaluation window or USE's command line. The latter is utilised in our BPMNChecker plug-in to retrieve object names and display them as error objects.







b

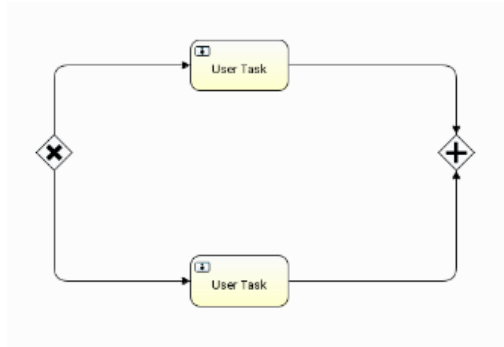


c

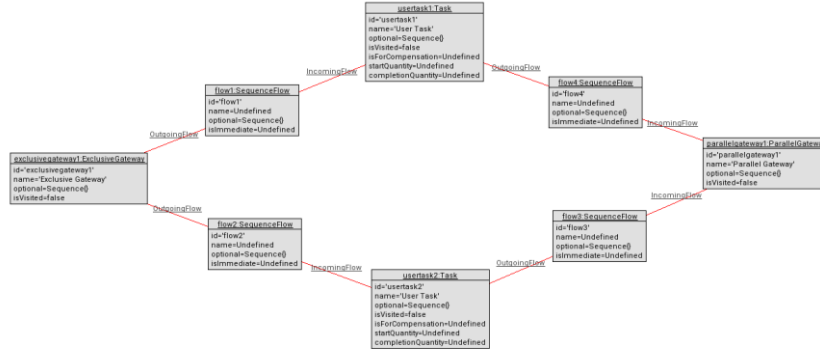
Figure 13. Visualisations of business process models in 4.5 (figure a), 4.4 (figure b) and 4.3 (figure c) as USE object diagrams. OCL query results are shown in red.

## 5.2. Discussion

Our algorithm for detecting deadlocks and multiple terminations requires a process to have at least one start event. Therefore, it cannot be applied on incomplete business process models.



a



b

Figure 14. A business process model containing no start event (a) and its corresponding object diagram (b).

The algorithm also assumes that the model satisfies the well-formedness invariants listed in the USE specification (no flow to start events and from end events, all nodes must be connected, gateways must split/fork or merge/join. If these constraints are not satisfied, it may fail.

Moreover, as we only implement a small subset of BPMN, elements such as lanes, pools, subprocesses and boundary events are not validated. We aim to cover the complete BPMN 2.0 specification in our future work.

## 6. Related work

While there are numerous tools handling the creation of business process models, they do not support validation and error-checking. As a result, a number of approaches have been proposed.

Awad and Puhlman [8] proposed a method using BPMN-Q to query for deadlock patterns. This approach has the advantage of low computational effort and user-friendliness, due to BPMN-Q being a visual query language. However, it is not applicable to implicit splits and joins. While we consider them anti-patterns, our deadlock detection algorithm still works with BPMN models containing them.

Approaches involving the transformation of BPMN models into other process models, such as process automata [3] or Petri nets [9] attempt to take advantage of extensive tooling support for them. On the other hand, extra steps must be taken when writing queries for deadlock patterns.

## **7. Conclusion and future work**

In this paper, we have presented a method to validate business process model with USE and OCL queries. Our approach involves using a plug-in to parse BPMN XML files and check for well-known anti-patterns, including deadlocks. It can be used to assist modellers through the visualisation of errors in the model.

Our method could be easily extended to support more BPMN elements and attributes, as well as detecting additional patterns. In the future, we will focus on expanding the method's coverage in order to be fully compatible with the latest BPMN specification. We also plan to investigate means to make the detection process more intuitive for end users – for example, visualise the errors in a model editor or suggesting/making necessary corrections for simple errors.

## References

- [1] A. C. e. Correia, Quality of Process Modeling Using BPMN: A Model-Driven Approach, 2014.
- [2] M. Sadowska, “An approach to assessing the quality of business process models expressed in BPMN,” *e-Informatica Software Engineering Journal*, vol. 9, no. 1, 2015.
- [3] N. Tantitharanukul, P. Sugunnasil and W. Jumpamule, “Detecting deadlock and multiple termination in BPMN model using process automata,” in *2010 International Conference on Electrical Engineering/Electronics Computer Telecommunications and Information Technology (ECTI-CON)*, 2010.
- [4] M. Gogolla, J. Bohling and R. Mark, “Validating UML and OCL models in USE by automatic snapshot generation,” *Software & Systems Modeling*, vol. 4, no. 4, p. 2005, 386-398.
- [5] F. Büttner and M. Gogolla, “Modular embedding of the object constraint language into a programming language,” in *Brazilian Symposium on Formal Methods*, 2011.
- [6] Object Management Group, “BPMN Semantic XSD,” [Online]. Available: <http://www.omg.org/spec/BPMN/20100501/Semantic.xsd>.
- [7] Camunda, “Camunda BPMN Model,” [Online]. Available: <https://github.com/camunda/camunda-bpmn-model>.
- [8] A. Awad and F. Puhlmann, “Structural detection of deadlocks in business process models,” in *International Conference on Business Information Systems*, 2008.
- [9] I. Raedts, M. Petkovic, Y. S. Usenko, J. M. E. van der Werf, J. F. Groote and L. J. Somers, “Transformation of BPMN Models for Behaviour Analysis,” *MSVVEIS*, vol. 2007, pp. 126-137, 2007.