A Formal Approach to Modelling and Verification of Business Process Collaborations

Barbara Re

November 29, 2018

Table of Contents

Basic Idea

OMG did not provide a rigorous semantics for BPMN 2.0

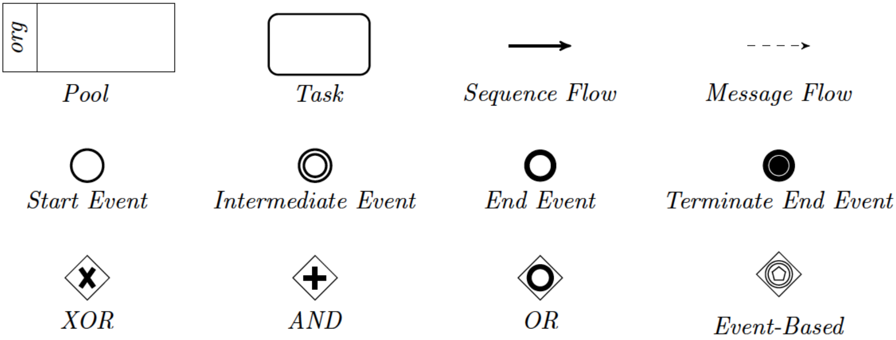
- **Possible miss-interpretations** due to the usage of the natural language in the specification of the standard
- Formal **verification** is not supported per se
- ...

Provide a **direct formalisation of BPMN 2.0 collaboration diagram** in terms of **Labelled Transition Systems**

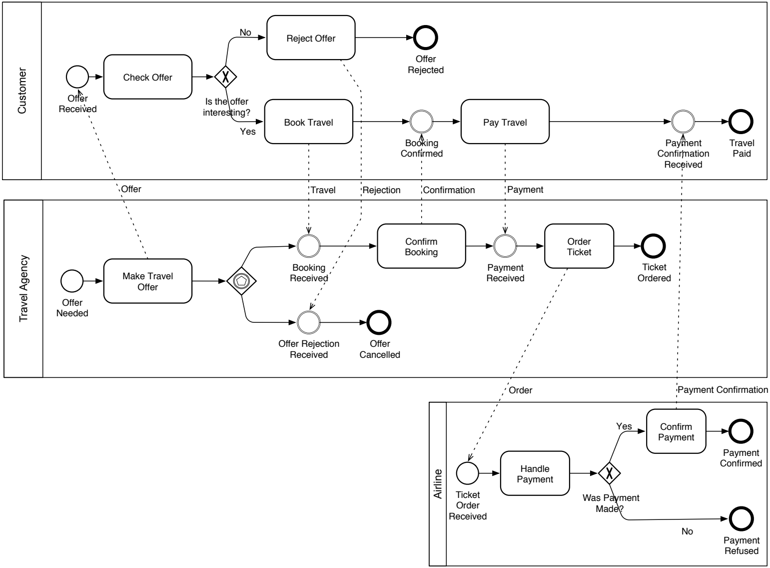
Major Benefits

- It is **a native semantics**, rather than a mapping to a formal notation equipped with its own semantics
- Besides core elements such as tasks, gateways, etc., it takes into account **message exchange**, and **termination events** which are often overlooked by other formalisations
- It is suitable to model business processes with **arbitrary topology**, without imposing restrictions to the modeler, such as well-structuredness

Considered BPMN 2.0 Elements



BPMN 2.0: Airline Collaboration Example



BNF Syntax Derivation

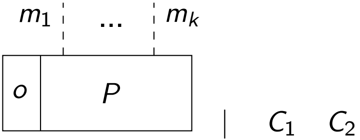
A term of the **syntax** can be straightforwardly **obtained from a BPMN model** by decomposing:

- Collaboration in collection of pools
- Processes in collections of nodes
- Edges in two parts

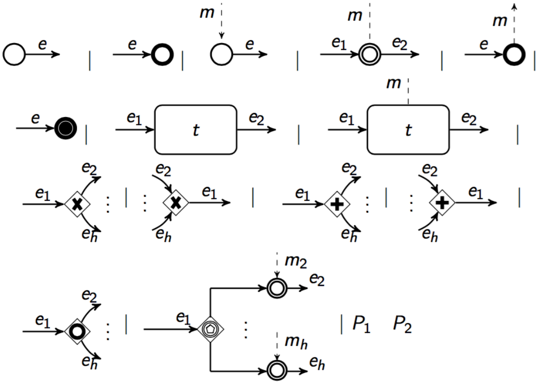
Considered specifications are **well-defined** (easily checked through static analysis), this implies that distinct pools, messages and sequences have **different names**

BNF Syntax

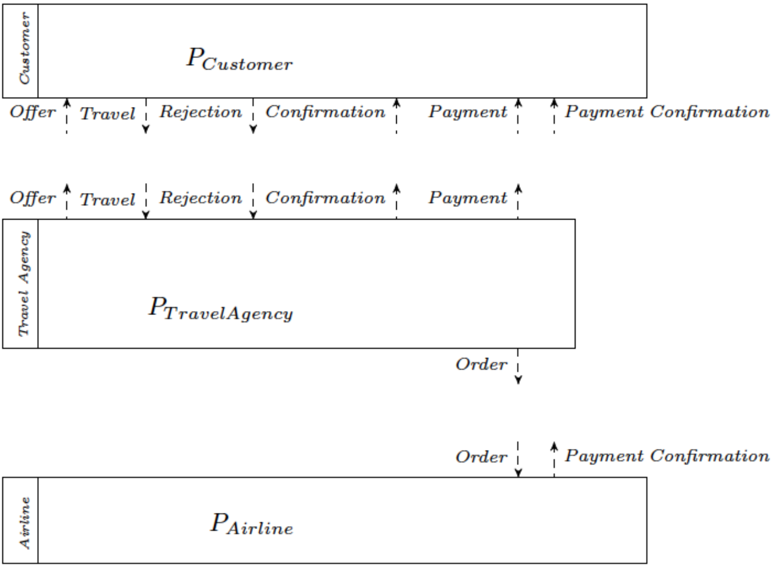
(*Collaborations*) $C ::=$



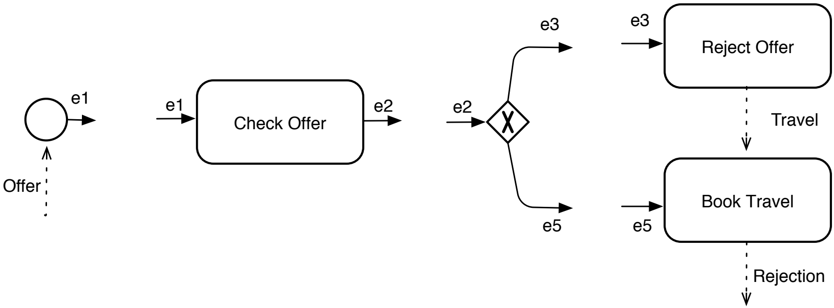
(*Processes*) $P ::=$



BNF Syntax: Airline Collaboration Example



BNF Syntax: Airline Collaboration Example (Customer)



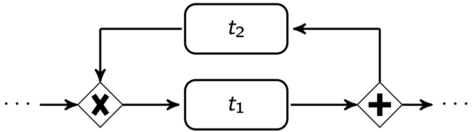
Operational Semantics: Marked Collaborations

The semantics of BPMN is given in terms of **marked collaborations**

A **marking** is a distribution of tokens (possibly multiple) over pool message edges and process elements that indicate message arrivals and the process nodes that are active or not in a given step of the execution

- A **single token** is denoted by $\bullet$
- **Multiple tokens** labelling a message edge m (resp. sequence edge e) are denoted by $m.n$ (resp. $e.n$), where $n \in \mathbb{N}$ is the token multiplicity

Even if **single instance business process** are considered **the use of tokens with multiplicity is necessary**



The **initial marking** of a collaboration assigns a single token to a start event of each process in the collaboration

Operational Semantics: Process Labels

The **labeled transition relation** of the LTS defining the semantics of **collaborations layer** and uses an auxiliary transition relation defining the semantics of **process layer**

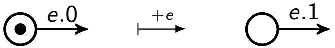
$$P \xmapsto{\alpha} P'$$

$$
\begin{array}{llllllllll}
(\mathit{Actions}) & \alpha & ::= & \tau & \mid & !m & \mid & ?m & & \\
(\mathit{Internal\ actions}) & \tau & ::= & \mathit{running}\ t \mid & & \mathit{completed}\ t \mid & & (-\tilde{e_1}, +\tilde{e_2}) \mid & & \mathit{kill}
\end{array}
$$

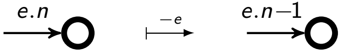
- τ denotes an action **internal** to the process
- $!m$ and $?m$ denote **sending** and **receiving** actions, respectively
- *running* t and *completed* t denote the **start** and **completion** of the execution of task t, respectively
- $(-\tilde{e_1}, +\tilde{e_2})$ denotes **movement of workflow tokens** in the process graph
- *kill* denotes the **termination** of the whole process

Operational Semantics Processes Layer: Events (I)

(*E-Start*)

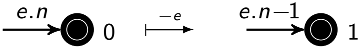


(*E-End*)



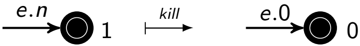
$n > 0$

(*E-Terminate*)



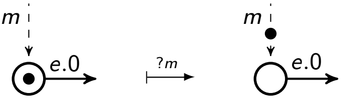
$n > 0$

(*E-Kill*)

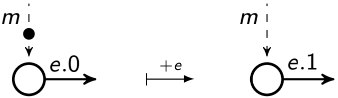


Operational Semantics Processes Layer: Events (II)

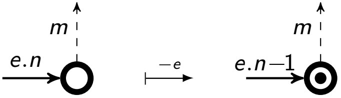
(*E-Receive*)



(*E-MsgStart*)

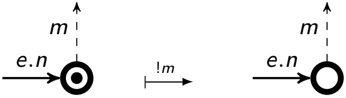


(*E-MsgEnd*)



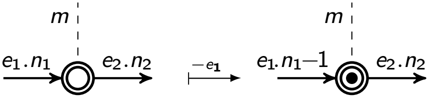
$n > 0$

(*E-Send*)



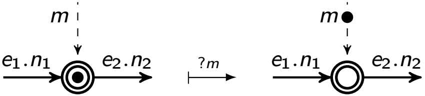
Operational Semantics Processes Layer: Events (III)

(E-*EnableInterm*)

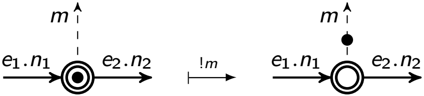


$n_1 > 0$

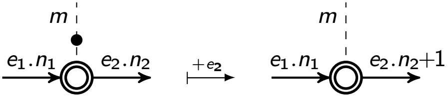
(E-*ReceiveInterm*)



(E-*SendInterm*)

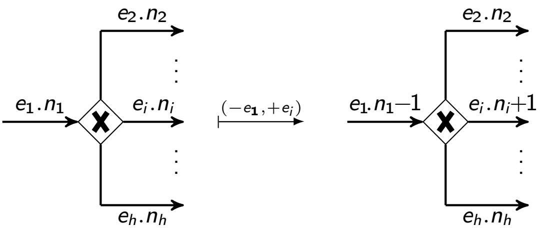


(E-*CompleteInterm*)



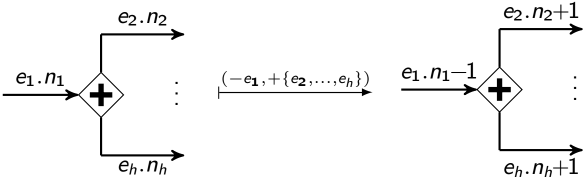
Operational Semantics Processes Layer: Gateway (Split)

(*G-XorSplit*)



$n_1 > 0$
$2 \leqslant i \leqslant h$

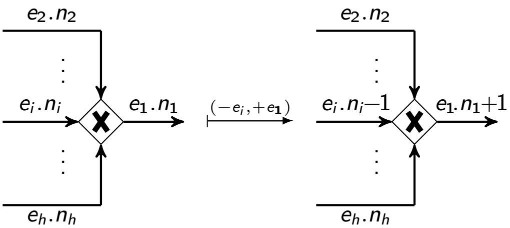
(*G-AndSplit*)



$n_1 > 0$

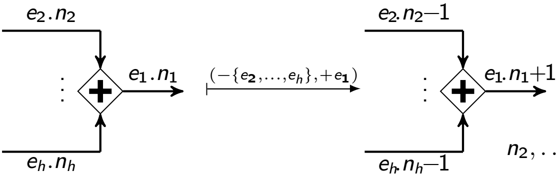
Operational Semantics Processes Layer: Gateway (Join)

(*G-XorJoin*)



$n_i > 0$
$2 \leqslant i \leqslant h$

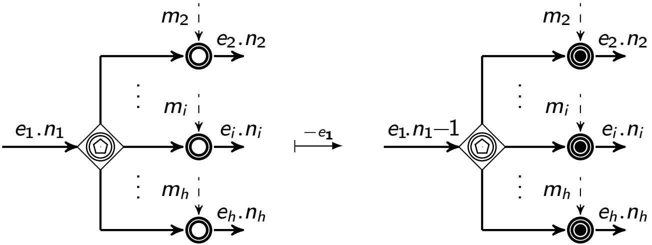
(*G-AndJoin*)



$n_2, \dots, n_h > 0$

Operational Semantics Processes Layer: Event Based Gateway (I)

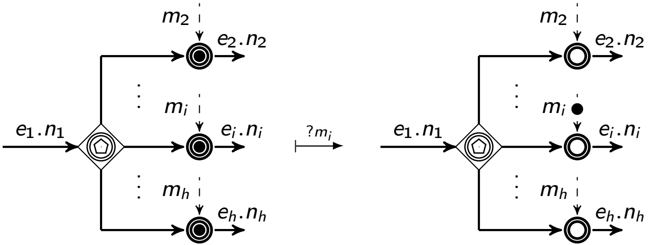
(*G-EnableEvent*)



$n_1 > 0$

Operational Semantics Processes Layer: Event Based Gateway (II)

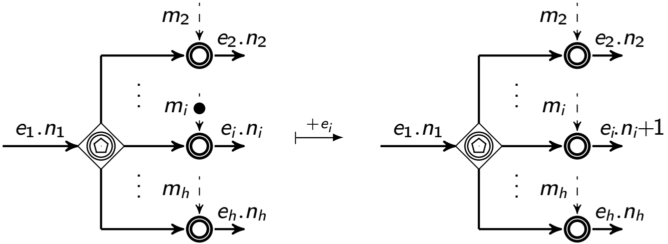
(*G-ReceiveEvent*)



$2 \leqslant i \leqslant h$

Operational Semantics Processes Layer: Event Based Gateway (III)

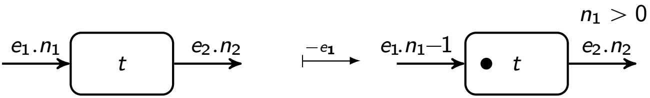
(*G-CompleteEvent*)



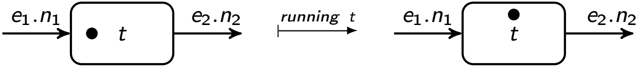
$2 \leqslant i \leqslant h$

Operational Semantics Processes Layer: Task

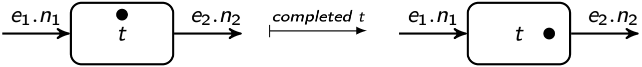
(*T-Enable*)



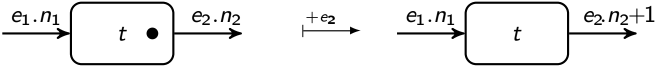
(*T-Running*)



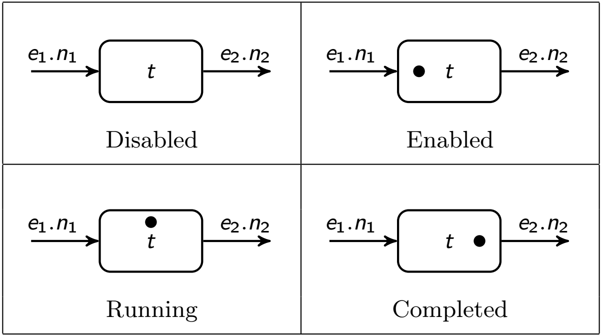
(*T-Complete*)



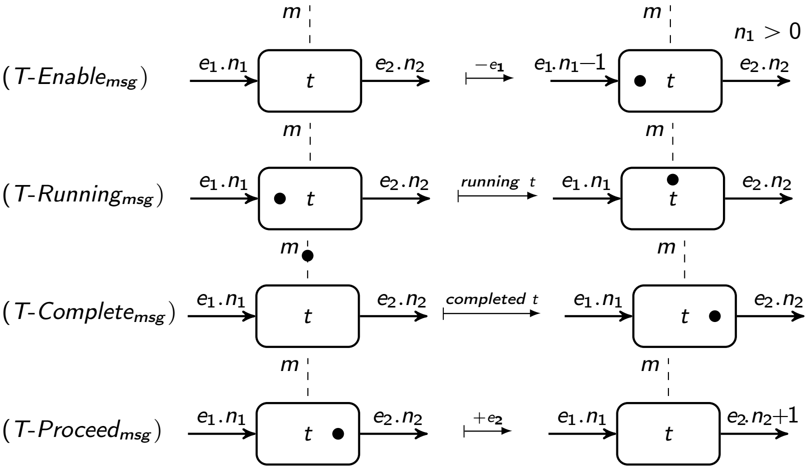
(*T-Proceed*)



Status of Non-Communicating Tasks

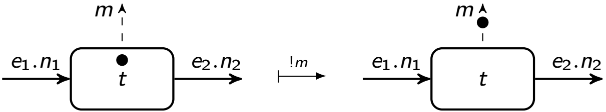


Operational Semantics Processes Layer: Communicating Task (I)

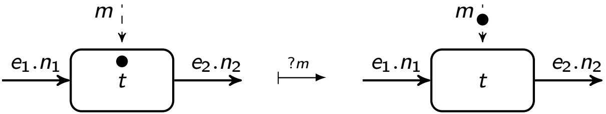


Operational Semantics Processes Layer: Communicating Task (II)

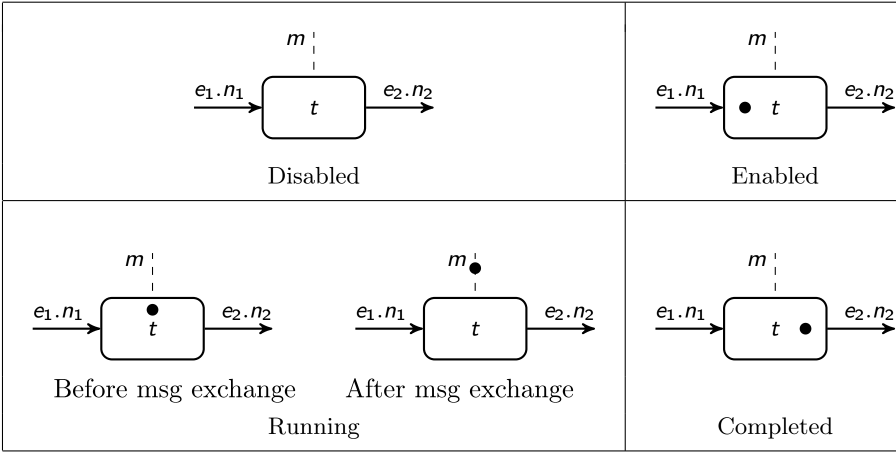
(*T-Send*)



(*T-Receive*)



Status of Communicating Tasks



Operational Semantics Process Layer: Node Collection

$$(\textit{N-MarkingUpd}) \qquad \frac{P_1 \xmapsto{(-\tilde{e}_{\mathbf{1}},+\tilde{e}_{\mathbf{2}})} P_1'}{P_1 \quad P_2 \xmapsto{(-\tilde{e}_{\mathbf{1}},+\tilde{e}_{\mathbf{2}})} P_1' \quad P_2 \cdot (-\tilde{e}_1, +\tilde{e}_2)}$$

$$(\textit{N-Kill}) \qquad \frac{P_1 \xmapsto{\textit{kill}} P_1'}{P_1 \quad P_2 \xmapsto{\textit{kill}} P_1' \quad P_2\dagger}$$

$$(\textit{N-Interleaving}) \qquad \frac{P_1 \xmapsto{\alpha} P_1' \qquad \alpha \notin \{(-\tilde{e}_1, +\tilde{e}_2), \textit{kill}\}}{P_1 \quad P_2 \xmapsto{\alpha} P_1' \quad P_2}$$

Operational Semantics: Making Updating Function

$$P \cdot (-\tilde{e_1}, +\tilde{e_2})$$

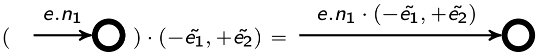
Marking updating function returns a process obtained from P by unmarking (resp. marking) edges in $\tilde{e_1}$ (resp. $\tilde{e_2}$)

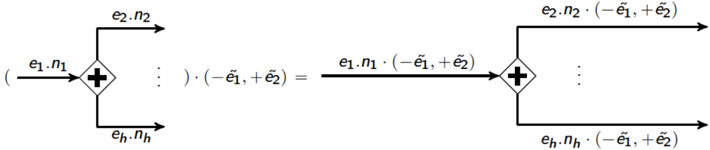
It is inductively defined on the structure of process P and, with abuse of notation, extends to terms of the form $e.n$ as follows:

$$e.n \cdot (-\tilde{e_1}, +\tilde{e_2}) \;=\; \begin{cases} e.n{-}1 & \text{if } e \in \tilde{e_1} \\ e.n{+}1 & \text{if } e \in \tilde{e_2} \\ e.n & \text{otherwise} \end{cases}$$

Few cases of the definition (the others are similar):

$$(P_1 \quad P_2) \cdot (-\tilde{e_1}, +\tilde{e_2}) = P_1 \cdot (-\tilde{e_1}, +\tilde{e_2}) \quad P_2 \cdot (-\tilde{e_1}, +\tilde{e_2})$$





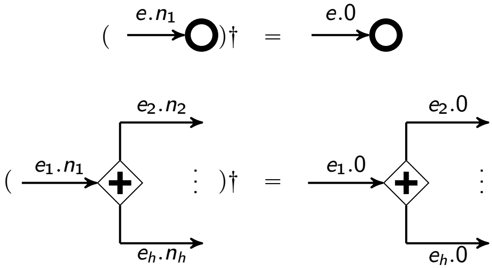
Operational Semantics: Killing Function

$$P\dagger$$

Killing function returns a process obtained from P by completely unmarking it

It is inductively defined on the structure of process P as follows (few cases):

$$(P_1 \quad P_2)\dagger \quad = \quad P_1\dagger \quad P_2\dagger$$



Operational Semantics: Collaboration Labels

$$C \xmapsto{l} C'$$

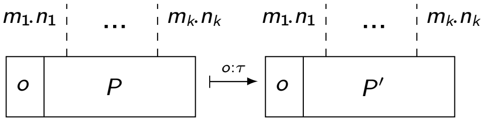
$$(Labels) \qquad l \quad ::= \quad o : \tau \quad | \quad o :?m \quad | \quad o_1 \rightarrow o_2 : m$$

- $o : \tau$ denotes an internal action **action** τ **peformed by the process instance** of organisation o
- $o :?m$ denotes an internal action **action** $?m$ **peformed by the process instance** of organisation o
- $o_1 \rightarrow o_2 : m$ denotes the **exchange of a message** m from organisation o_1 to o_2

Operational Semantics Collaboration Layer (I)

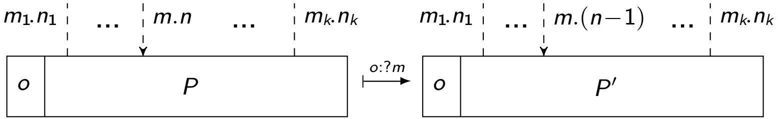
$(C\text{-}Internal)$

$$P \xmapsto{\tau} P'$$



$(C\text{-}Receive)$

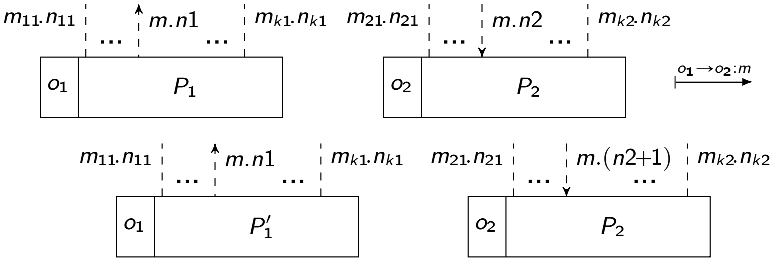
$$P \xmapsto{?m} P' \qquad n > 0$$



Operational Semantics Collaboration Layer (II)

(*C-Deliver*)

$$P_1 \xmapsto{!m} P_1'$$

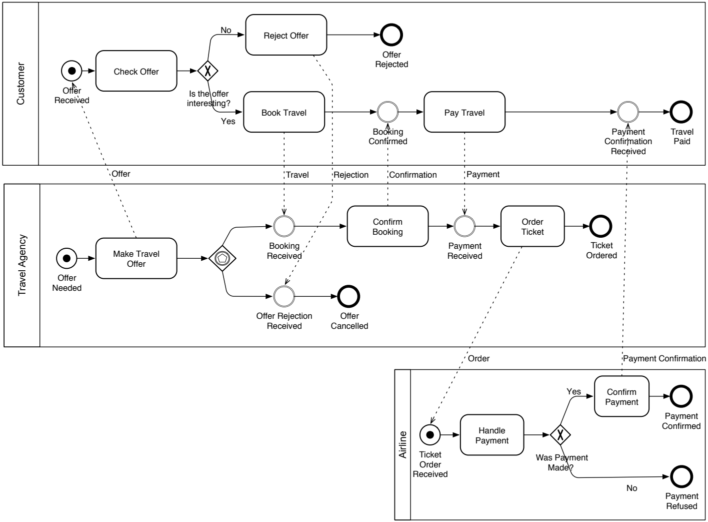


(*C-Interleaving*)

$$\frac{C_1 \xmapsto{l} C_1'}{C_1 \quad C_2 \xmapsto{l} C_1' \quad C_2}$$

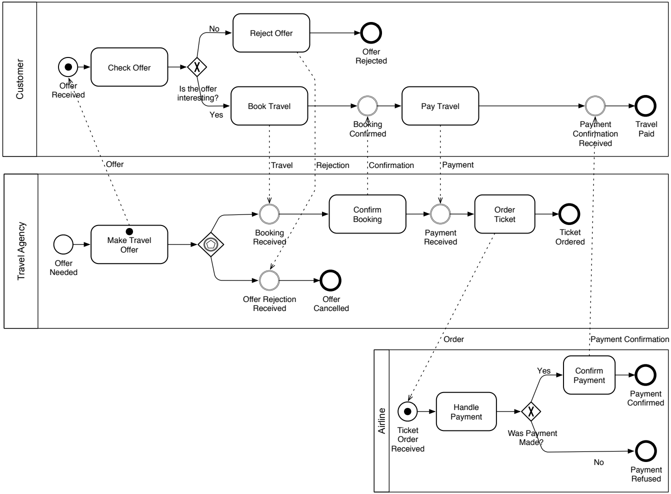
Semantics of the Running Example (I)

Initial Configuration



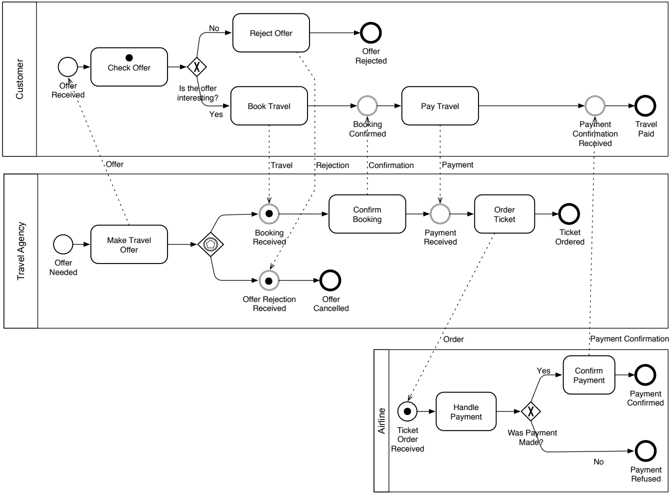
Semantics of the Running Example (II)

Make Travel Offer Task is Running (before message exchange) in the Travel Agency Pool, while Customer and Airline are in the Initial State



Semantics of the Running Example (III)

Check Offer Task is Running in the Customer Pool, Event Based is Enable in the Travel Agency Pool, while the Airline is in the Initial State



Verification - Soundness & Safeness Properties

The investigation of **properties of business process models** is an important aspect of business process management

Behavioral correctness relates to potential sequences of execution as defined by the process model

The business process model must be **analyzed** and **improved** to make sure

- If it actually includes **all desired instances**
- If a certain **desired** property at the business process model level can be shown, then all process instances based on that business process model expose this property
- If a certain property is **undesired** at the business process model level can be shown, then all process instances do not have to expose this property

Verification is an **error-prone activities**, to be repeated several times, for which automatic tools are necessary

Verification - Safeness, Soundness & Compliance

Safeness refers to the occurrence of no more than one token at the same time along the same sequence edge of a process instance

Soundness property can be encoded in terms of three simpler ones:

- **Option To Complete**, requiring that a process instance can always complete, once started
- **Proper Completion**, requiring that there exists no running or enabled activity for this instance when the process instance completes
- **No Dead Activities**, requiring that a process model does not contain any dead activity, i.e., for each activity there exists at least one producible trace which contains the activity

Business Process Compliance means the execution of business processes in compliance with imposed rules

> These properties naturally extend to process collaborations, requiring that the process instances of all involved organisations satisfy them

Properties Formalization in LTL

In the verification we use Linear temporal logic (LTL) such as a modal temporal logic with modalities referring to time

In LTL, one can encode formulae about the future of paths, e.g., a condition will eventually be true, a condition will be true until another fact becomes true, etc.

The formulas are obtained as composition of the following basic cases:

- <> ϕ, where the operator <> (corresponding to the LTL operator F) is used to verify if a formula ϕ **eventually** holds. That is, in any possible execution path we always encounter a state where ϕ holds
- [] ϕ, where the operator [] (corresponding to the LTL operator G) is used to verify if a formula ϕ **globally** holds. That is, ϕ holds in all states encountered in any possible execution path
- ϕ -> φ, where the operator -> is the standard boolean implication

Properties Abstraction via Sub-formulae

We also define sub-formulae which make the approach effective because they abstract low level parts of LTL formulae and will be reused whatever is the model to analyze

aBPstarts. It is satisfied if at least a process in the collaboration can start

aBPstartsParameterized. It is satisfied if the process associated to a specific Pool (identified by its Organization Name) can always start

aTaskComplete. It is satisfied if in the all Collaboration, there is at least one Task that can always be marked as "completed"

aTaskCompletedParameterized. It is satisfied if a specified Task in the Collaboration will always complete

aTaskRunningParameterized. It is satisfied if a specified Task in the Collaboration will be able to run

aBPSndMsg (resp. **aBPRcvMsg**). It is satisfied if a specified message is sent (resp. received)

Properties Verification - Safeness

Safeness can be encoded in terms of one single condition only, *safeState*

safeState evaluates to true in states that satisfy the auxiliary function *noMultipleToken*, which verifies that on each sequence edge there is at most one token

```
[] safeState(poolName)
```

The above formula verifies that from any state ([]) it is true or not

Properties Verification - Soundness Option To Complete

Option To Complete requirs that a process instance can always complete, once started

```
[](aBPstarts(poolName) -> <>aBPends(poolName))
```

The above formula verifies that from any state (`[]`) in which the pool can start (`aBPstarts(poolName)`), we eventually reach a state (`<>`) where the pool completes its execution (`aBPends(poolName)`)

Properties Verification - Soundness Proper Completion

Proper Completion requiring that there exists no running or enabled activity for this instance when the process instance completes

In particular, we check that whenever a token reaches the end of the pool, then no other token remains unused within the pool

```
[](aBPends(poolName) -> NoDandlingToken(poolName))
```

Differently from Option To Complete, now the right-hand side of the implication does not have a <> operator because we check the *NoDanglingToken* condition on the same state that satisfied *aPoolEnds*

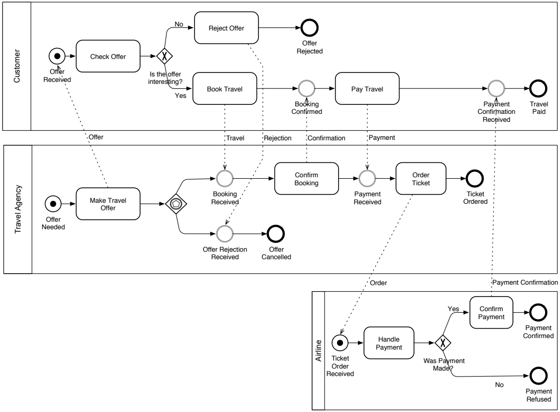
Properties Verification - Soundness No Dead Activities

No Dead Activities relies that a process model does not contain any dead activity, i.e., for each activity there exists at least one producible trace which contains the activity

```
<> aTaskRunning(taskName)
```

It relies on the verification of the condition *aTaskRunning*, which establishes that a given task can be set, at least once, in the status Running (meaning that the task is currently being executed)

Compliance Rules - Example




```
aTaskCompletedParameterized("Handle Payment")
|-> aTaskCompletedParameterized("Confirm Payment")
```

false

Checking Properties on our Running Example

1	<>[] safeState(poolName)	true
2	<>[](aPoolCanStart(poolName) |-> <>aPoolEnds(poolName))	false
3	<>[](aPoolEnds(poolName) |-> NoDandlingToken(poolName))	true
4	<><> aTaskRunning(taskName)	True
5	<>aBPoolstarts	true
6	<>aBPstartsParameterized("Airline")	false
7	<>aBPoolends	true
8	<>aBPendsParameterized("Customer")	false
9	<>aTaskComplete	true
10	<>aTaskComplete("Handle Payment")	false
11	[] (aTaskRunningParameterized("Confirmation Booking") ->(<>aTaskCompleteParameterized("Confirmation Booking")))	true
12	aTaskCompleteParameterized("Handle Payment") |-> aTaskCompleteParameterized("Confirm Payment")	false
13	aBPoolSndMsg("Customer", "Payment") |-> aBPoolRcvMsg("Customer","Payment Confirmation")	false

Maude Implementation

To concretely **validate our theoretical definitions** and to practically **enable verification of BPMN collaborations**, both the **syntax** and the **operational semantics have been implemented using Maude**[1]

Maude enables formal verification of BPMN collaborations, e.g., by means of the MAUDE state space generator and the MAUDE LTL model checker

Using our Maude implementation of BPMN we can **verify some properties by expressing them in terms of LTL formulae** and by using the Maude LTL model checker

[1] http://pros.unicam.it/tools/bprove

Maude Operation (Syntax)

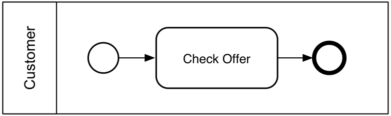
Each BPMN element is declared as a **Maude operation** written according to the following general form

```
op element(_, ...  ,_) :  Sort-1 ...  Sort-k -> RSort .
```

Its meaning is:

- the keyword op indicates the definition of an operation
- `element` is the name of the operation that we define
- `(_, ...  ,_)` specifies that the `element` operation is characterized by a number of parameters, whose sorts `Sort-1` , ... , `Sort-k` are reported after the `:` symbol
- the `->` symbol is followed by the resulting sort `RSort` of the `element` operation
- finally, the `.` symbol is used to end the line of code

Minimal Collaboration Example




```
collaboration(
 pool( "Customer" ,
    proc(
     {emptyAction}
     start( enabled , "e1" .  0 ) |
     task( disabled , "e1" .  0 , "o1" .  0, "Check Offer" ) |
     end( "o1" .  0 )
    ) , in:  emptyMsgSet , out:  emptyMsgSet
  )
) .
```

Minimal Collaboration Example Syntax - Start

```
collaboration(
 pool( "Customer" ,
    proc(
     {emptyAction}
     start( enabled , "e1" .  0 ) |
     task( disabled , "e1" .  0 , "o1" .  0, "Check Offer" ) |
     end( "o1" .  0 )
    ) , in:  emptyMsgSet , out:  emptyMsgSet
  )
) .
```

```
op start(_,_) :  Status Edge -> ProcElement .
ops disabled enabled running completed ...  :  -> Status .
op _._ :  EdgeName EdgeToken -> Edge .
```

Minimal Collaboration Example Syntax - Task

```
collaboration(
 pool( "Customer" ,
    proc(
     {emptyAction}
     start( enabled , "e1" .  0 ) |
     task( disabled , "e1" .  0 , "o1" .  0, "Check Offer" ) |
     end( "o1" .  0 )
    ) , in:  emptyMsgSet , out:  emptyMsgSet
  )
) .
```

```
op task(_,_,_,_) :  Status Edge Edge TaskName -> ProcElement .
ops disabled enabled running completed ...  :  -> Status .
op _._ :  EdgeName EdgeToken -> Edge .
```

Minimal Collaboration Example Syntax - End

```
collaboration(
 pool( "Customer" ,
    proc(
     {emptyAction}
     start( enabled , "e1" .  0 ) |
     task( disabled , "e1" .  0 , "o1" .  0, "Check Offer" ) |
     end( "o1" .  0 )
    ) , in:  emptyMsgSet , out:  emptyMsgSet
  )
) .
```

```
op end(_) :  Edge -> ProcElement .
op _._ :  EdgeName EdgeToken -> Edge .
```

Minimal Collaboration Example Syntax

```
collaboration(
 pool( "Customer" ,
   proc(
    {emptyAction}
    start( enabled , "e1" .  0 ) |
    task( disabled , "e1" .  0 , "o1" .  0, "Check Offer" ) |
    end( "o1" .  0 )
   ) , in:  emptyMsgSet , out:  emptyMsgSet
 )
) .
```

```
op proc(_) :  ActProcElement -> Process .
op __ :  Action ProcElement -> ActProcElement .
op pool(_,_,in:_,out:_) :  OrgName Process Msgs Msgs -> Collaboration .
op collaboration(_) :  ActCollaboration -> Model .
op __ :  CollaborationAction Collaboration -> ActCollaboration .
```

Maude Rewriting Rules (Semantics)

According to the presented syntax, we implemented the semantics by means of **rewriting rules** and **conditional rewriting rules**, which we write respectively in the following general forms

```
rl [Label] :  Term-1 => Term-2 .

crl [Label] :  Term-1 => Term-2 if Condition-1 /\...  /\Condition-N .
```

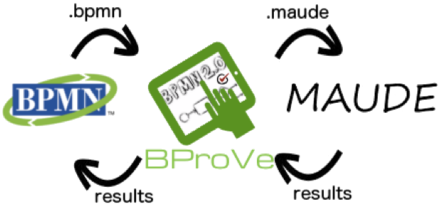
Some Rewriting Rules

```
rl [E-Start] :
start( enabled , IEName .  IEToken )
=>
{tUpd(emptyEdgeSet , IEName .  IEToken)}
start( disabled , IEName .  increaseToken( IEToken ) ) .

crl [N-Interleaving] :
ProcElem1 | ProcElem2
=>
{Action1} (ProcElem1’ | ProcElem2 )
if ProcElem1 => Action1ProcElem1’ /\ isInterleaving(Action1) .

crl [C-Internal] :
pool(OrgName1, proc( {Action1} ProcElem1), in:inMsgSet , out:outMsgSet)
=>
{collab(OrgName1 , Action1’)}
pool(OrgName1, proc({Action1’}ProcElem1’), in:inMsgSet , out:outMsgSet)
if ProcElem1 => {Action1’} ProcElem1’ / \ isInternal(Action1’) .
```

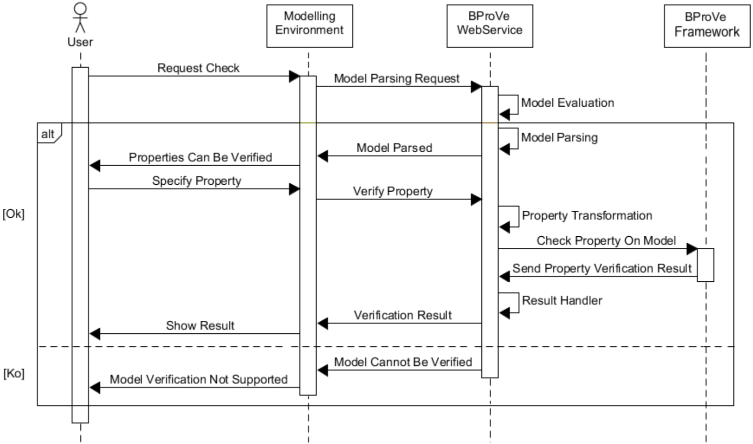
Eclipse and Apromore Tool Chain



A tool chain integrating the verification environment with BPMN modelling environments, such as Eclipse BPMN Modeller and Apromore platform has been developed

Verification will be applied in a **wide range of real scenarios** for a more extensive evaluation of the approach

BProVe - Sequence Diagram



Conclusions

> The lack of a shared, well-established, comprehensive formal semantics for BPMN was the main driver of our work

A **direct formalisation of BPMN 2.0 collaboration diagram** in terms of **Label Transition System** was defined

- It enables designers to freely specify their processes with an **arbitrary topology** supporting the adherence to the standard, without the requirement of defining **well-structured** models

The semantics was implemented in **Maude**

- This enables the exploration of the evolution of **BPMN collaborations**, and it permits to exploit the **analysis tool set** provided by **Maude**

A **complete tool chain** presenting a modelling environment and a service for the **automatic verification of properties over the designed BPMN models** was designed and implemented

References

Corradini, F., Polini, A., Re, B., & Tiezzi, F. (2015, October). An Operational Semantics of BPMN Collaboration. In International Conference on Formal Aspects of Component Software (pp. 161-180). Springer International Publishing.

F. Corradini, F. Fornari, A. Polini, B. Re, A. Vandin, F. Tiezzi. BProVe: Tool Support for Business Process Verification. 32nd IEEE/ACM International Conference on Automated Software Engineering, ASE 2017 - Tool Demo. Urbana Champaign, Illinois, USA, October 30 - November 3, 2017.

F. Corradini, F. Fornari, A. Polini, B. Re, A. Vandin, F. Tiezzi. BProVe: a Formal Verification Framework for Business Process Models. 32nd IEEE/ACM International Conference on Automated Software Engineering, ASE 2017. Urbana Champaign, Illinois, USA, October 30 - November 3, 2017.

Flavio Corradini, Fabrizio Fornari, Andrea Polini, Barbara Re, and Francesco Tiezzi. A Formal Approach to Modelling and Verification of BPMN Collaboration. Science of Computer Programming, Vol. 166, pp. 35 - 70, June 2018.

Fabrizio Fornari, Marcello La Rosa, Andrea Polini, Barbara Re and Francesco Tiezzi. Checking Business Process Correctness in Apromore. CAiSE 2018 FORUM - Information Systems in the Big Data Era. Springer, LNBIP, vol. 317, pp. 114-123. Tallinn, Estonia, 11-15 June 2018.