Transformation of BPMN Diagrams to YAWL Nets

JianHong YE
School of Computer Science and Technology
Huaqiao University, Quanzhou Fujian 362021, China
Email: leafever@163.com
Wen SONG
School of Mathematics and Computer Engineering
Xihua University, Chengdu Sichuan 610039, China
Email: yl1505@163.com

Abstract—Business Process Modeling Notation (BPMN) is the de facto standard for modeling business processes on a conceptual level. However, BPMN lacks a formal semantics and many of its features need to be further interpret, Consequently that hinders BPMN as a standard to statically check the semantic correctness of models. YAWL (Yet Another Workflow Language) allows the specification of executable workflow models. A transformation between these two languages enables the integration of different levels of abstraction in process modeling. This paper discusses how to transform BPMN diagrams to YAWL nets. The benefits of the transformation are threefold. Firstly, it clarifies the semantics of BPMN via a mapping to YAWL. Secondly, the deployment of BPMN business process models is simplified. Thirdly, BPMN models can be analyzed with YAWL verification tools.

Index Terms—BPMN, YAWL, Transformation, Algorithm

I. INTRODUCTION

Process modeling is used at different levels of abstraction. First, models serve to communicate as-is business processes, pinpoint improvement options, conduct resource and cost analysis and capture to-be processes. The Business Process Modeling Notation (BPMN) [1] is the de facto standard for process modeling at this level. On the other hand languages targeting at technically realizing business processes is used as input for process execution engines. The YAWL [2] is a standard for implementing process-oriented composition of web services, with a strictly defined execution semantics, a first-class concept of "task", and sophisticated support for data mappings and task-to-resource allocation.

BPMN is a graph-oriented language in which controlling nodes can be connected in arbitrary way. It primarily targets at domain analysts and is supported by many modeling tools. The notation inherits and combines elements from a number of previously proposed notations for business process modeling, including the XML Process Definition Language (XPDL) [3] and the Activity Diagrams component of the Unified Modeling Notation (UML) [4].

BPMN provides a number of advantages to modeling business processes, it offers a process flow modeling technique that is more conducive to the way of business analysts model and its solid mathematical foundation is expressly designed to map to business execution languages. BPMN is already supported by more than 54 tools (see www.bpmn.org), but in its current form, BPMN lacks the semantic precision which is required to capture fully executable business processes. Consistent with the level of abstraction targeted by BPMN, none of these tools support the execution of BPMN models directly. Close inspection of existing translation from BPMN to BPEL standard, the one sketched in [1], [5]–[7], shows these translations fail to fulfill the key requirements, such as completeness, automation, readability, etc. The translation patterns and algorithms in these papers address issues that arise generally when translating from graph-oriented process languages to block-structured ones. However, mapping between graph-oriented and block-structured process definition languages is notoriously challenging, it is likely to require refinement as well as testing and debugging, which defeat the purpose of BPEL as a domain-specific language. Another attempt at defining a formal semantics for a subset of BPMN did so using Petri nets [8]–[11]. The proposed mapping serves not only the purpose of disambiguating the core constructs of BPMN, it also provides a foundation to statically check the semantic correctness of BPMN models. However, their semantics does not properly model multiple instances, exception handling, message flows and OR-join.

YAWL is a workflow language specially designed to support the 20 workflow patterns [12] that proposed by Van der Aalst, ter Hofstede, Kiepuszewski and Barros in an intuitive manner. YAWL can be used as a lingua franca to express the behavior of Web services (for example, described using BPEL or OWL-S [13]). YAWL has a well defined formal semantics. Furthermore, the basis on Petri nets provides a firm tool for the formal analysis of real-world services. In order to benefit from the expressive power of YAWL, a large amount of business process development has been done in the form of tools and plug-ins.

Manuscript received Jun 11, 2009; revised Oct 13, 2009; accepted Oct 19, 2009.
This work was supported in part by the Science Foundation of Huaqiao University grants 09BS314 and the Special Foundation for Young Scientists of Fujian Province.

© 2010 ACADEMY PUBLISHER
doi:10.4304/jsw.5.4.396-404
models are mapping to YAWL, such as the Event-driven Process Chain (EPC) to YAWL [14] and BPEL to YAWL [15].

The transformation from BPMN to YAWL can be used as an instrument to implement process-oriented applications. It also opens the possibility of reusing static analysis techniques available for YAWL. Like Petri nets, YAWL has a formally defined semantics that enables the analysis of YAWL nets to detect semantic errors such as deadlocks. Close inspection of existing translations from BPMN to YAWL, e.g. sketched in [16], however, reveals several properties and assignments are missing in the mapping, the translations fail to fulfill the following key requirements: (i) message flows are lost; (ii) transactions and compensation handlers are not covered; (iii) complex control-flow structures. However, mapping BPMN to one between BPMN and BPEL, especially with regards to gateways are neglected. At first glance, this mapping may seem straightforward. Indeed, the conceptual mismatch between BPMN and YAWL is not as significant as the one between BPMN and BPEL, especially with regards to control-flow structures. However, mapping BPMN to YAWL turns out to be tricky in the details, revealing subtle differences between the two languages.

Our goal is to provide a methodology for transforming a model from BPMN to YAWL. The mapping is in the following five ways:

- Although BPMN and YAWL share most of their concepts, there is a fundamental difference in the way of joins and splits are treated in each language. While BPMN inherits the connector types from EPCs which define them as first-class objects independent of functions, YAWL includes joins and splits in task objects. Accordingly, there is no direct equivalent in YAWL elements for BPMN connector chains, i.e. multiple consecutive connectors.

- YAWL requires processes to have only one start and one end condition. In BPMN, multiple start and end events are allowed.

- BPMN task or subprocess has a lot of attributes, which attributes can be applied in YAWL and which ones should be extended.

- A message flow is used to show the transmission of messages between two participants via communication actions such as send task, receive task, or message event in different pools in BPMN, although the information of pool or lane will be lost in conversion. How to map these messages to flow messages and not affect the YAWL initial marking is a challenge.

- BPMN exception handling is captured by exception flow. The conversion should be clear regarding the semantics of an exception handler attached on a task or subprocess.

The rest of the paper is organized as follows. Section II provides the mathematical notations. Our contribution starts in Section III, which illustrates the solutions of the transformation from BPMN to YAWL. Section IV presents the structure of the tool implementation and its application to static analysis of BPMN models. Finally, Section V concludes and outlines future work.

II. BASIC DEFINITIONS

A BPMN process, which uses the core subset of BPMN elements as shown in Figure 1, is referred to as a core BPMN process. We define the syntax of the core BPD.

**Definition 1 (Core BPMN Process):**

A core BPMN process is a tuple: $M = (O, A, ε, p, L, T, φ, S, T_{R}, O_{S}, O_{e}, ε_{I}, ε_{O}, ε_{E}, ε_{M}, ε_{ε},

- $O$ is a set of objects which can be partitioned into disjoint sets of activities $A$, events $ε$, gateways $g$.

- $A$ can be partitioned into disjoint sets of tasks $T$ and subprocesses invocation activities $S$.

- $P/L$ is a set of pools/lanes. For any node $o ∈ O$, $P(o)/L(o)$ is a function showing which pool/lane the object $o$ belongs to.

- $T_{R} ⊆ T$ is a set of receiving tasks.

- $ε$ can be partitioned into disjoint sets of start events $ε_{S}$, intermediate events $ε_{I}$ and events $ε_{E}$.

- $ε_{I}$ can be further partitioned into disjoint sets of intermediate events without any trigger $ε_{I}^{ε}$, intermediate message events $ε_{I}^{m}$, intermediate timer events $ε_{I}^{tr}$, intermediate exception events $ε_{I}^{e}$, and intermediate cancel events $ε_{I}^{c}$.

- $g$ can be partitioned into disjoint sets of parallel fork gateways $g_{A}$, parallel join gateways $g_{A_{j}}$, based $XOR$ decision gateways $g_{X}$, $XOR$ merge gateways $g_{X^∗}$, event-based $XOR$ decision gateway $g_{X}$, inclusive decision gateways $g_{O}$, and inclusive merge gateway $g_{O_{i}}$.

- $F ⊆ O × O$ is the control flow relation, i.e. a set of sequence flows connecting objects.

- $Cond:F ∩ (g^{X \times} × O) → C$ is a function which maps sequence flows emanating from data-based $XOR$ gateways to conditions.$^{1}$

- $Exc:ε_{I} → A$ is a function assigning an intermediate event to an activity such that the occurrence of the event signals an exception and thus interrupts the performance of the activity.

- $Mes \subseteq O_{l} × O_{j}, P(o') ≠ P(o''), o' ∈ O_{l} ∧ o'' ∈ O_{j}$, i.e. objects belong to different pools.

Definition 1 allows for graphs which are unconnected, not have start or end events, contain objects without any input and output, etc. Therefore we need to restrict the definition to well-formed core BPMN. Before this, we first define the predecessor and successor nodes.

**Definition 2 (Predecessor and Successor Nodes):** Let $O$ be a set of objects and $Edge$ be a binary relation over $O$, $Edge = F ∪ Cond ∪ Exc$. For each node $o$, we define the set of predecessor nodes $^{o}o = \{x ∈ O | (x, o) ∈ Edge ∧ o ∈ O \}$, and the set of successor nodes $^oO = \{x ∈ O | (o, x) ∈ Edge ∧ o ∈ O \}$.

$^{1}$A condition is a Boolean function operating over a set of propositional variables that can be abstracted out of the control flow definition. The condition may evaluate to true or false, which determines whether or not the associated sequence flow is taken during the process execution.
Definition 3 (Well-formed core BPMN): [17] A core BPMN process $P$ in Definition 1 is well formed if and only if relation $F$ satisfies the following requirements:

- $\forall s \in E^s \cup dom(Exc), s^* = \phi \land |s^*| = 1$, i.e. start events and exception events have an in-degree of zero and an out-degree of one,
- $\forall e \in E^e, e^* = \phi \land |e^*| = 1$, i.e. end events have an out-degree of zero and an in-degree of one,
- $\forall x \in A \cup (E^l \setminus dom(Exc)), |x^*| = 1$ and $|x^*| = 1$, i.e. activities and non-exception intermediate events have an in-degree of one and an out-degree of one, respectively,
- $\forall g \in g^X \cup g^X \cup g^O : \{g^* \} = 1 \land |g^*| > 1$, i.e. fork or decision gateways have an in-degree of one and an out-degree of more than one,
- $\forall g \in g^X \cup g^X \cup g^O : \{g^* \} > 1 \land |g^*| = 1$, i.e. join or merge gateways have an out-degree of one and an in-degree of more than one,
- $\forall g \in g^X, g^* \subseteq E^I_x \cup E^T_x \cup T^R$, i.e. event-based XOR decision gateways must be followed by intermediate message or timer events or receive tasks,
- $\forall g \in g^X, \exists x \in g^*, Convl((g, x)) = \neg \land_{x \in g^* \setminus \{x\}} = Convl((g, y))$, i.e. $(g, x)$ is the default flow among all the outgoing flows from $g$,
- $\forall e \in E^e, sF^S_F \land xF^e_F$, i.e. every object is on the path from a start event or an exception event to an end event.

Definition 4 (YAWL-Net): [2] A YAWL-net $N$ is defined as a tuple $(C, i, o, T, Flow, split, join, rem, nofi)$ such that:

- $C$ is a set of conditions,
- $i \in C$ is the unique input condition,
- $o \in C$ is the unique output condition,
- $T$ is a set of tasks,
- $Flow \subseteq (C \setminus \{o\} \times T) \cup (T \times C \setminus \{i\}) \cup T \times T$ is the flow relation.

- split : $T \rightarrow \{AND, XOR, OR\}$ specifies the split behavior of each task,
- join : $T \rightarrow \{AND, XOR, OR\}$ specifies the join behavior of each task,
- rem : $T \rightarrow \rho(T \setminus \{i, o\})$ specifies the token to be removed from the tasks and conditions given in the mapping,
- nofi: $T \rightarrow N^inf \times N^inf \times \{dynamic, static\}$ specifies the multiplicity of each task (minimum, maximum, threshold for continuation, and dynamic/static creation of instances).

III. FROM BPMN TO YAWL

We only consider map well-formed core BPMN diagrams to YAWL nets in this paper, using simplified notation $M = (O, P, L, F, Exc, Mes)$ for their representation. The study is focused on control-flow constructs.

A. Activities, Events, Gateways, Sequence flow and Message Flow

Figure 2 gives the transformation from a set of BPMN activities, events, gateways and sequence flows. An intermediate event or task is mapped to an atomic task with one input and output. The other activities except for Ad-hoc subprocess are mapped onto corresponding composite task or multiple instance tasks with one predecessor and one successor. BPMN start and end events are easy to transform if there is only one. In this case, the BPMN start event maps to YAWL input condition and the end event to a output condition. Multiple start and end events will be discussed in the Section III-B.

As gateways are independent elements in BPMN, which is allowed to build so-called gateway chains, i.e. paths of two or more consecutive gateways. Splits and joins in YAWL are only allowed as part of tasks. As a consequence, there may be need to introduce empty
Each activity will be performed one at a time.

Algorithm 1: B_Mes2Y_Flow(Mes m)

Input: ε, γ, m ∈ Mes, m = (x, γ)[P(x) ≠ P(y)]. Suppose there exists an input task t_{inTask} and output task t_{outTask, split} = OR join(t_{outTask}) = OR;

Output: yawl flow;

1. if ∀x ∈ ε, 3y ∈ ε ∪ T ∪ S, s.t. (x, y) ∈ Mes then
2. add empty task t’ with join(t’') = XOR to yawl;
3. add f ∈ Flow ∧ f = (t”, t_{outTask}) to yawl;
4. add f ∈ Flow ∧ f = (t’, y) to yawl;
5. if x ∈ S ∪ T ∧ |x| = 1 then
6. add f ∈ Flow ∧ f = (x, t’) ∧ split(x) = AND to yawl;
7. end
8. else if x ∈ S ∪ T ∧ |x| = 0 then
9. add f ∈ Flow ∧ f = (x, t’) to yawl;
10. end
11. else if x ∈ ε then
12. add f ∈ Flow ∧ f = (*x, t’) to yawl, delete x;
13. end
14. delete x;
15. end
16. if ∀x ∈ ε, 3y ∈ ε ∪ T ∪ S, s.t. (x, y) ∈ Mes then
17. add empty task t’ with split(t’’) = AND to yawl;
18. add f ∈ Flow ∧ f = (t’’, t_{outTask}) to yawl;
19. add f ∈ Flow ∧ f = (t’, y) to yawl;
20. if x ∈ S ∪ T ∧ |y| = 1 then
21. add f ∈ Flow ∧ f = (t’, y) ∧ join(y) = AND to yawl;
22. end
23. else if x ∈ S ∪ T ∧ |y| = 0 then
24. add f ∈ Flow ∧ f = (t’, y) to yawl;
25. end
26. else if y ∈ ε then
27. add f ∈ Flow ∧ f = (t’, y) to yawl, delete y;
28. end
29. delete y;
30. end
31. while |Mes| ≠ 0 do
32. if x ∈ (T ∪ S) ∧ y ∈ (T ∪ S) then
33. if |x| = 0 ∧ |y| = 0 then
34. add f ∈ Flow ∧ f = (x, y) to yawl;
35. end
36. else if |x| = 1 ∧ |y| = 0 then
37. add f ∈ Flow ∧ f = (x, y) ∧ split(x) = AND to yawl;
38. end
39. else if |x| = 0 ∧ |y| = 1 then
40. add f ∈ Flow ∧ f = (x, y) ∧ join(y) = AND to yawl;
41. end
42. else if |x| = 1 ∧ |y| = 1 then
43. add f ∈ Flow ∧ f = (x, y) ∧ split(x) = AND ∧ join(y) = AND to yawl;
44. end
45. end
46. else if x ∈ (T ∪ S) ∧ y ∈ ε then
47. if |x| = 0 then
48. add f ∈ Flow ∧ f = (x, y) to yawl;
49. end
50. else if |x| = 1 then
51. add f ∈ Flow ∧ f = (x, y) ∧ split(x) = AND to yawl;
52. end
53. delete ε;
54. end
55. else if x ∈ ε ∧ y ∈ (T ∪ S) then
56. if |y| = 0 then
57. add f ∈ Flow ∧ f = (*x, y) to yawl;
58. end
59. else if |y| = 1 then
60. add f ∈ Flow ∧ f = (*x, y) ∧ join(y) = AND to yawl;
61. end
62. delete ε;
63. end
64. else if x ∈ ε ∧ y ∈ ε then
65. add f ∈ Flow ∧ f = (*x, y) to yawl, delete ε and ε;
66. end
67. end

C. Activities Macro and Ad-hoc construct

An activity in a BPD can have attributes specifying its additional behavior, these attributes can be set to determine the values of nolf in the corresponding task in YAWL. Table I demonstrated how to map the BPMN attributes to the values of nolf.

There is another special attribute for Ad-hoc subprocess activity. This attribute defines if the activities within the process can be performed in parallel or must be performed sequentially. Without lose the generality, the multiple instance attributes for activities in Ad-hoc subprocess can referred from Table I. The example in Figure 4 considered each activity will be performed one at a time.

Figure 3. Transformation of multiple start and end events
such as receive task, user task, service task etc. refer to hand, some detail discussions involving activity attributes between them do not include message flow. On the other predecessor or successor of nodes and the binary relation for mapping the message flow. Figure 5 shows some distinguish these cases. Algorithm 1 gives the method of AdhocOrdering attribute set to Sequential ... attribute set to Parallel 
AdhocCompletionCondition set to include the completion of Task A and B

<table>
<thead>
<tr>
<th>BPMN activity attributes</th>
<th>YAWL task nofi</th>
</tr>
</thead>
<tbody>
<tr>
<td>LoopType</td>
<td>LoopMax</td>
</tr>
<tr>
<td>standard</td>
<td>m</td>
</tr>
<tr>
<td>standard</td>
<td>m</td>
</tr>
<tr>
<td>LoopType</td>
<td>LoopMax</td>
</tr>
<tr>
<td>multiInstance</td>
<td>m</td>
</tr>
<tr>
<td>multiInstance</td>
<td>m</td>
</tr>
<tr>
<td>multiInstance</td>
<td>m</td>
</tr>
<tr>
<td>multiInstance</td>
<td>dynamic</td>
</tr>
</tbody>
</table>

Note that the $n$, $m$ is integer and the expression $xq_i$ (i = 1, 2, 3) is evaluated at run time to help determine how many instances of activities are to be created. The LoopCondition and MICondition will be replaced in a mathematical expression to be either tested as True or False or to be evaluated to update the value of Properties in actual use.

### E. Exception Handling

While the intermediate event is attached to the boundary of an activity, either a task or a subprocess, they create an exception flow. The event will respond to specific triggers to interrupt the activity and redirect the flow through the intermediate event. The source of the trigger may from external or caused by a “throw” intermediate event from any other active in the process. YAWL presents a direct and intuitive support of the remove tokens, sounds like “vacuum cleaner” removing tokens from selected parts of a net. At the same time, we have to impose one restriction that a subprocess associated with exception handing is not allowed to be interrupted by the occurrence of the exception event in itself, which will violate the seal principle in a subprocess. This is ambiguous in the BPMN specification states. We will describe two different source of the trigger circumstances in the following:

Firstly, assuming the Timer, Message, Exception and Error trigger will be invoked from the external of the process execution, Figure 6(1) shows the mapping of an exception associated with a task or subprocess via an exception task and cancel arc in YAWL.

Secondly, except the Timer trigger, an intermediate event attached on a task or subprocess will be invoked by an event occurs, and location in the process with the name exact consistency. Assuming there are two intermediate events with the same name, if ambiguity is possible, we use throw or catch as subscript convenient for marking their positions in the process. For example, ithrow is an intermediate exception event in the process, and icatch refer to an intermediate event associated with a task invoking exception handing activities by $i_{\text{throw}}$. Figure 6(2) depicts the mapping. Algorithm 2 shows the exception flow conversion algorithm.
(1) mutual process instantiation dependency

(2) Task(subprocess) to task(subprocess)

(3) Task(subprocess) to start event

(4) End event to task(subprocess)

(5) End event to start event

Figure 5. Examples of BPMN message flow to YAWL flow

(1) Trigger is external of the process

(2) Trigger is caused by a “throw” intermediate event in the process

Figure 6. Mapping of exception flow

Figure 7. Reduction of connector chains events

Algorithm 2: B_Exc2Y_Flow(Exc e)

Input: e ∈ Exc = (x, y) | x ∈ ε, y ∈ O, r is the Task(Subprocess) associated with x on the boundary;
Output: yawl flow;
1 if Name(x) is not exclusive, i.e.
2 ∃z ∈ ε ∧ |z| ≠ φ ∧ Name(x) ≡ Name(z) then
3 map z to a task;
4 add cancel edge(r, z) to yawl;
5 map z to a task and Split(z) = AND;
6 add f ∈ Flow ∧ f = (z, x) to yawl;
7 else
8 map x to a task;
9 add cancel edge(r, x) to yawl;
10 end

F. Transforming gateway chains

As joins and splits are first class elements of BPMN while in YAWL they are part of tasks. As a consequence, there may be need to introduce empty tasks only to map a connector. This is in particular the case with connector chains. Figure 7(1) illustrates how a connector chain is transformed. If a join connector is followed by a split, they can be combined into one empty task. Otherwise, splits and joins can be combined with the pre-event predecessor function or the post-event successor function, respectively.

In addition, if there have a split connector as the successor of one task (subprocess), which can be embedded into the task(subprocess). It is similar with a task (subprocess)’s predecessor. Figure 7(2) and (3) respectively show these possible.

G. Transformation Algorithm and Example

We traverse the BPMN process graph and take advantage of the fact that YAWL does not enforce an alternation of tasks and conditions. Basically, we ignore events except start or end events. Therefore, most states of the generated YAWL process model are associated with implicit conditions.

Based on the mapping of each of the components aforementioned, we now define an algorithm to translate a well-formed core BPD into YAWL. The algorithm is arranged in two stages, the first stage is depending on the type of the current node, its predecessor nodes and successor nodes, respective elements of the YAWL target model are generated. The second stage is reducing gateway chains if necessary. These steps repeated until no elements can be folded.

Now, we can analyze the complexity of the algorithm 3. Given a well-formed BPD M = (O, P, L, F, Exc, Mes),
### Algorithm 3: WF\_BPMN2YAWL (BPD $M$)

Input: a well-formed BPD $M = (O, P, F, L, E_{exc}, M_{es})$

Output: a YAWL model $Y = (C, i, o, T, Flow, split, join, rem, nolfs)$

1. Add YAWL input and output conditions: $i \in C$ and $o \in C$.
2. For each task $t \in T$:
   1. If $|x(t)| > 1$ then
      1. Add OR split task $t_{inTask}$.
   2. If $|x(t)| > 1$ then
      1. Add OR join task $t_{outTask}$.
   3. End
3. While $|M| \neq 0$
   1. For each node $n \in O$: if node is not be mapped; $M = M \setminus n$
   2. If node is an event $\in E_{exc}$ and node is not be mapped; $M = M \setminus n$
   3. If node is a flow $\in Flow(n)$ then
      1. Add $c \in C$ and $n = t_{inTask}$ (or $n = t_{outTask}$).
   4. End
   5. End
   6. End

For each transformation of BPMN node, the conversion operation is performed in linear time and the algorithm will be completed. The completeness is guaranteed because BPMN are coherent: every node will be processed exactly once (no influence by the exception handling event). Assuming that converse a BPMN node is processed exactly once (no influence by the exception handling event). The completeness is granted because the extreme case, the algorithm will terminate when each element has been processed or a time-out has occurred, the result needs to be archived (task $archive$), and in parallel, if the complaint evaluation has been completed, the actual processing of the complaint (task $process\_complaint$) can start. Next, the processing of the complaint is checked via task check. If the check result is not ok, the complaint requires re-processing. Otherwise, if the check result is ok and also the questionnaire has been archived, a notice will be sent to inform the complainant about the completion of the complaint handling (task $send\_notice$).

![Figure 8](http://www.ilog.com/products/jviews/diagrammer/bpmnmodeler/)

Figure 8. Structure of the BPMN to YAWL transformation.

### IV. Empirical Evaluation

As a proof of concept we have implemented the algorithm, an open-source plug-in called BPMN2YAWL is available in ProM 5.0. Each reader interested in this field can use the http://www.processmining.org web page for a more complete overview and download the latest version. Figure 9 shows the structure of the tool that we implemented.

We use the ILog BPMN Modeler\(^2\) as a graphical editor to create BPMN models. To the author’s knowledge, no existing tool for modeling BPMN can perform properties checks. We can analyze the BPMN model after them into well-formed BPMN models, how to generate a well BPD can be archived. The BPMN models into a YAWL net and export the YAWL net as a XML file. The XML file can serve as input to a YAWL-based verification tool \([11]\), \([18]\). A lot of properties such as the deaklock\_free, no dead\_task, proper completion, no OR\_join and soundness \([19]\)–\([21]\) can be checked via these tools.

We tested BPMN2YAWL on a set of models\(^3\), some collected from the BPMN Web log\(^2\), and the others are designed by the authors. Table II shows the size of each tested BPMN models in terms of number of tasks, events, gateways, subprocesses and message flows. It also shows the size of the resulting YAWL-nets in terms of number of conditions, atomic tasks and composite tasks.

![Figure 9](http://www.ilog.com/products/jviews/diagrammer/bpmnmodeler/)

**Figure 9. Structure of the BPMN to YAWL transformation.**

\(^2\)http://www.ilog.com/products/jviews/diagrammer/bpmnmodeler/

\(^3\)This set of test models are included in the distribution of the BPMN2YAWL tool, http://is.tue.nl/trac/prom/wiki/TestBPMN
TABLE II.

RESULTS OF IMPLEMENTING THE MAPPING TOOL TO EXISTING MODELS

<table>
<thead>
<tr>
<th>Model No</th>
<th>BPMN Model</th>
<th>YAWL net</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>tasks</td>
<td>events</td>
</tr>
<tr>
<td>1</td>
<td>6</td>
<td>5</td>
</tr>
<tr>
<td>2</td>
<td>20</td>
<td>10</td>
</tr>
<tr>
<td>3</td>
<td>20</td>
<td>4</td>
</tr>
<tr>
<td>4</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>5</td>
<td>2</td>
<td>4</td>
</tr>
</tbody>
</table>

Fig 10 shows a screenshot of ProM, generating the YAWL net from the No.1 BPMN model, which violate the deadlock-free and soundness properties via the analysis of YAWL editor⁴. Moreover, We detected model 2 contained incomplete process executions. Models 0, 3, 4 and 5 did not contain any errors.

V. CONCLUSION AND FUTURE WORK

Ongoing work aims at extending the BPMN2YAWL plugin in order to make the transformation reversible. After generating a model, the plugin will be able to propagate changes in the YAWL net into the BPMN diagram (and vice-versa) in order to maintain the models synchronized. We aim to analyze the whole BPMN process models with YAWL verification tools such as WofYAWL [11]. This will provide insight into the correctness of large enterprise models described in BPMN. We also plan to investigate process mining using BPMN processes and focusing on the OR-join semantic particularities of BPMN.

⁴http://www.yawl-system.com/
ACKNOWLEDGMENT

The authors wish to thank W.M.P. van der Aalst, Remco M. Dijkman and Xun Pu for their careful reading of an earlier version of this paper.

REFERENCES


JianHong Ye was born in Xiamen, China. He received his PhD degree in computer software and theory from University of Electronic Science and Technology of China in 2009, his MS degree in computer software and theory from the Xihua University in 2006, and his BS degree in computer science from the Fuzhou University in 2002.

He is currently a lecturer of computer science at Huaqiao University, China. He was a recipient of the Fujian Province National Science Foundation CAREER award.

Wen Song was born in Chengdu, China. He received his MS degree in computer software and theory from the Guizhou University in 1991.

He is currently professor of computer science at Xihua University, China. His current research interests include theory of Petri nets and mathematical logic.

Prof. Wen Song is a senior member of China Computer Federation and a member of Petri net Special Commission. He is a recipient of the China National Science Foundation CAREER award.