

BPMN 2.0 Execution Semantics Formalized as Graph Rewrite Rules

Remco Dijkman and Pieter Van Gorp

Eindhoven University of Technology, The Netherlands
{r.m.dijkman|p.m.e.v.gorp}@tue.nl

Abstract. This paper presents a formalization of a subset of the BPMN 2.0 execution semantics in terms of graph rewrite rules. The formalization is supported by graph rewrite tools and implemented in one of these tools, called GrGen. The benefit of formalizing the execution semantics by means of graph rewrite rules is that there is a strong relation between the execution semantics rules that are informally specified in the BPMN 2.0 standard and their formalization. This makes it easy to validate the formalization. Having a formalized and implemented execution semantics supports simulation, animation and execution of BPMN 2.0 models. In particular this paper explains how to use the formal execution semantics to verify workflow engines and service orchestration and choreography engines that use BPMN 2.0 for modeling the processes that they execute.

1 Introduction

The Business Process Modeling Notation (BPMN) version 2.0 [1] has a well defined execution semantics. The execution semantics can be used by tool vendors to develop tools for simulation, animation and execution of business process models.

Tool developers must strictly adhere to the execution semantics, because interchange of business process models between tools is only possible if both syntax *and* semantics are preserved. In particular business process analyst can model a business process in one tool and expect the modeled process to behave in a certain way because they are familiar with the execution semantics of BPMN and/or because the modeling tool supports animation of the modeled process. If the modeled business process is subsequently exported to another tool, such as a workflow engine, the business process analyst expects the workflow engine to behave in the exact same way. The interchangeability of business process model semantics has become even more important now that interchange of syntax is facilitated well by the XML Process Definition Language (XPDL) [2], which is being adopted by an increasing number of tool vendors [3], making it more and more likely that process analysis use multiple tools during the business process lifecycle.

To further facilitate interchange of business process model semantics test suites are required to test whether a tool conforms to a specified execution

semantics. The use of test suites to test whether a tool conforms to a specified syntax is already common. For example, the XPDL specification contains a Document Type Definition (DTD) that can be used to verify whether the XPDL output that is generated by a tool conforms to the XPDL standard.

Therefore, the goal of this paper is to present a formalized execution semantics for a subset of BPMN. The execution semantics is defined using graph rewrite rules [4], which facilitate a strong traceability between BPMN execution rules and the formal semantics, thus enabling a formalized semantics that can easily be validated. As a proof of concept, the execution semantics is implemented in a graph rewrite tool called GrGen [5]. The formalized execution semantics is intended to be used as a test suite, to test conformance of tools that implement the BPMN execution semantics, in particular workflow engines and service choreography and orchestration engines. Therefore, this paper also presents an architecture to enable conformance testing of implementations of the BPMN execution semantics.

The remainder of this paper is structured as follows. Section 2 explains the concept of graph rewrite rules, which is used to formalize the BPMN execution semantics in this paper. Section 3 presents the formalization of the execution semantics. Section 4 presents a tool architecture that can be used, in combination with the execution semantics, to do conformance testing of tools that implement the BPMN execution semantics. Section 5 presents related work on defining the BPMN semantics and section 6 concludes.

2 Graph Rewrite Rules

Until the early 1970s, the semantics of programming languages was defined using tree-based structures. This followed from the use of context-free grammars for defining their textual syntax. Since diagrammatic languages did not always contain a clear tree basis, new techniques were clearly needed (as explained further by [6]).

In the meanwhile, significant theoretical and practical progress has been made and a dedicated conference (ICGT) is approaching its fifth edition. Various theories (such as the so-called double-pushout approach [7]) are actively being used and extended to reason about increasingly powerful language constructs for defining rewrite rules. On the practical side, the community holds a yearly contest involving various case studies for comparing the power of the supportive tools. The previous edition of that contest has indicated that graph transformation tools can successfully be used for re-implementing existing BPMN related transformation programs in a more concise manner [8].

In this paper, we describe how we are using one of the contest's most successful graph transformation tools on a new BPMN related case study. Although we do not aim to explain each individual GrGen language construct, this section should enable BPM researchers without previous experience with graph transformation to comprehend some fragments of our formalization and to explore further details by executing the related rewrite system in debugging mode. To

facilitate the latter, we provide an online virtual machine containing all software related to this paper [9].

A graph rewrite rule is the basic building block of any graph transformation system. It consists of two conceptual parts: its left-hand side specifies a pattern defines the precondition for rule application whereas its right-hand side defines the pattern that should be realized when *applying* the rule on a match in the host graph. Graph rewrite rules can be specified both in a textual and a graphical manner. Figure 1 shows an example of a graph transformation rule. The rule is shown both using graphs and using the BPMN notation. The left-hand side and the right-hand side of the rule are shown on the left-hand and right-hand of the large arrow. The rule matches a graph in which there exists a node ‘sf’ of the type ‘SequenceFlow’ from which there is an edge named ‘t’ of type ‘Tokens’ to a node ‘tok’ of the type ‘Token’ and an edge of type ‘To’ to a node ‘a’ of type ‘Activity’. The rule realizes the deletion of the edge named ‘t’ and the creation of an edge of type ‘Tokens’ from ‘a’ to ‘tok’. In other words, if there is an activity with an incoming sequence flow with a token on it, the rule can move the token with it. Below, the same information is represented textually.

```

1 rule enterNodeOneTrans {
2   tok:Token <-ts:Tokens- sf:SequenceFlow --To
3     --> a:Activity;
4   modify{
5     delete(ts); a --:Tokens-> tok;
6 }

```

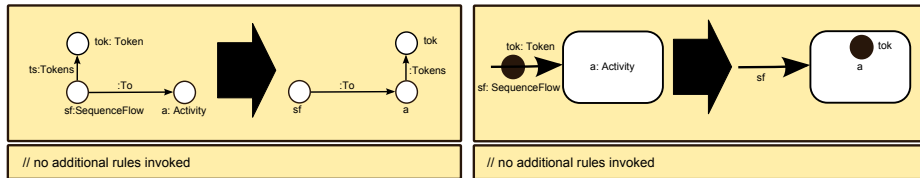


Fig. 1. Example of graphical representation of graph transformation rules

The GrGen pattern definition part (left-hand side) consists of node and edge declarations (or references to already declared elements). Nodes are declared by $n:t$, where n is an optional node identifier, and t its type. An edge e with type t and between nodes $n1$ and $n2$ is declared by $n1 -e:t-> n2$ (or $n1 <-e- n2$, or $n1 -e- n2$), whereas $n1 --> n2$ describes an anonymous edge. Nodes can also be left anonymous in patterns. The pattern “ $a:Activity --> . --> . --> . --> a$ ” for example describes a circular structure connected to activity node a without mentioning irrelevant names for the intermediate node or edge variables.

All nodes and edges are implicit instances of **Node** and **Edge** respectively. Gr-Gen also offers a textual language for defining more specific node types (such as **Activity** in the above example) and edge types. Both nodes and edges can have attributes and edges do not need to be directed (as opposed to term rewriting

or ECORE based approaches [10]). Nodes and edges are referenced outside their declaration by `n` and `-e->`, respectively.

The GrGen rewrite part (right-hand side) is specified by a block that is nested within the rule. In graph transformation theory, elements that are part of the right-hand side but no part of the left-hand side will be created upon rule application; elements that are part of the left-hand side but no part of the right-hand side will then be deleted. Although GrGen has a dedicated operator for this specification style, it also enables a variant where the deletion of elements needs to be encoded explicitly. In this paper, we only apply this variant. In the GrGen textual syntax, the rewrite part is then contained in a `modify` block. The GrGen user manual explains further details and also clarifies how patterns can be defined recursively, how matches can be executed in parallel, etc.

3 Formalization of the BPMN Execution Semantics

This section shows first how the rules for the BPMN execution semantics are controlled and then clarifies the meaning of some illustrative rules. As in the specification of the BPMN execution semantics, these rules are defined in terms of enabling and firing of elements, based on a token-game. The last subsection of presents the precise rules for moving tokens as well as keeping track of past markings. The complete execution semantics can be downloaded¹.

3.1 Execution Semantics Control

The “master script” of the GrGen implementation of the BPMN execution semantics is shown below.

```

1 include ../tests/Subprocess4.grbpmn
2 validate strict
3 xgrs [mapTo_SequenceFlows] | [mapTo_Associations] |
    addActivitiesToWFProcessWhereNeeded | [
    addSequenceFlowToWFProcessOfFromActivity]
4 xgrs [AddTokenForGlobalProcess] &&
    CreateInitialMarking
5 validate strict
6 debug xgrs (
7     enterNodeOneTrans $||
8     enterIntermediateEvent $||
9     enterExclusiveGateway $||
10    enterInclusiveGateway $||
11    enterParallelGateway $||
12    enterSubprocess $||
13    leaveNodeOneTrans $||
14    leaveSubprocessNormal $||
15    leaveExclusiveGateway $||
16    leaveParallelGateway $||
17    leaveInclusiveGateway
18 )[*]
```

Lines 6 to 18 execute the BPMN execution semantics rules. As documented in the rule names, some rules realize the *entry* of tokens in activity nodes whereas other rules ensure that tokens can leave such nodes again. The `xgrs` construct enables the execution of rewrite rules. These rules can be controlled by operators such as `[]` (parallel rule application), `||` (sequential OR with lazy evaluation), etc. The `$` operator flags on lines 6 to 18 indicates that the rules under consideration commute (they can be evaluated in any order). The `*` flag, which is used in `(rules)[*]`, instructs that the *rules* should be executed iteratively as long as matches are found.

¹ implementation available at: <http://is.tm.tue.nl/staff/rdijkman/bpmn.html>

Lines 1 through to 5 execute some initialization. Line 1 loads the input BPMN model and line 2 verifies whether the input graph conforms to the BPMN meta-model. Such syntactical checks have been quite useful during the concurrent development of the XPD L parser and the actual GrGen transformation. Line 3 transforms the input model in a form that is required to reason precisely about the state during BPMN model execution. More specifically, BPMN sequence flows and associations are transformed into pairs of edges with an intermediate node. This node can then be associated with the tokens that reside on that edge during process execution. Notice that this would not be necessary if GrGen would support so-called hyper-edges but we are in favor of GrGen’s simple yet sufficiently powerful metamodeling foundation.

Line 4 creates the initial marking (i.e., the initial overall process state). It puts tokens in the appropriate activities and creates a process instance element for each top-level workflow process.

3.2 Execution Semantics Rules

In this subsection we explain the actual execution semantics rules that are listed in lines 6 to 18.

The first rule ‘enterNodeOneTrans’ represents the execution semantics rule for an activity with a single incoming sequence flow. It is graphically represented in figure 2 and listed below.

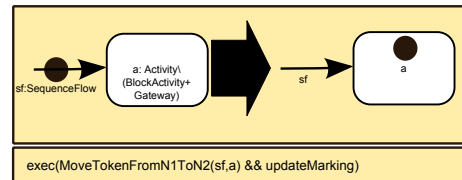


Fig. 2. Visual representation of rule *enterNodeOneTrans*

The left-hand side of the rule matches an element ‘a’ from the set of activities minus the set of block activities (i.e. embedded or referenced subprocesses) and gateways. The matched activity must have an incoming sequence flow ‘sf’ on which there is a token ‘tok’. Note that this means that an activity can be a task, an intermediate event or an end event, although figure 2 shows a task. Also note that this rule does not require that an activity only has one incoming sequence flow; it also applies in case an activity has multiple incoming sequence flows and, therefore, implements the rule that an activity with multiple incoming sequence flows behaves as an exclusive join gateway. If such a match is found, the rule executes the right-hand side, which involves the execution of the rule that moves the token ‘tok’ from one place to another (in this case from the sequence flow ‘sf’ to the activity ‘a’). Our implementation of the execution semantics stores the execution trace that was performed, this is done by executing the

‘updateMarking’ rule. The ‘MoveTokenFromN1ToN2’ and ‘updateMarking’ rules are used by all other rules. They are explained in subsection 3.3.

The second rule ‘enterIntermediateEvent’ represents the execution semantics rule for canceling a task or a subprocess, because an intermediate boundary event occurs. It is graphically represented in figure 3². The BPMN 2.0 standard also allows for intermediate boundary events that do not cancel the activity to which they are attached. However, the current version of XPDL does not support the ‘cancelActivity’ flag. Therefore, we do not consider that flag and use intermediate activities that cancel the activity to which it is attached as standard behavior, because this was the standard behavior for previous versions of BPMN.

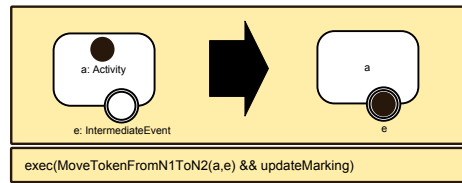


Fig. 3. Visual representation of rule *enterIntermediateEvent*

The third and fourth rule, ‘enterExclusiveGateway’ and ‘enterParallelGateway’, represent the execution semantics rules for multiple incoming sequence flows into an exclusive or parallel gateway, respectively. They are graphically represented in figure 4. The ‘enterParallelGateway’ rule uses a red box around a graph element. This red box represents that a graph is matched only if it *does not have* the construct in the red box. In this case the matched graph *does not have* a sequence flow to the gateway that *does not have* a token on it (i.e.: all sequence flows to the gateway must have a token on them).

Note that these execution semantics rules for the exclusive and parallel gateways also apply in case those gateways have multiple incoming sequence flows. Also note that, instead of executing the ‘MoveTokenFromN1ToN2’ rule, the parallel gateway executes the ‘MoveTokenFromIncomingToN2’ rule, because the tokens from all its incoming sequence flows must be moved instead of from a single incoming flow.

The fifth rule ‘enterInclusiveGateway’ represents the execution semantics rule for multiple incoming sequence flows into an inclusive gateway. It is graphically represented in figure 5. According to the BPMN specification, an “inclusive gateway is activated if:

- At least one incoming sequence flow has at least one Token and
- for each empty incoming sequence flow, there is no Token in the graph anywhere upstream of this sequence flow, i.e., there is no directed path (formed by Sequence Flow) from a Token to this sequence flow unless

² For the remaining rules, we only show the graphical representation. The listing can be downloaded.

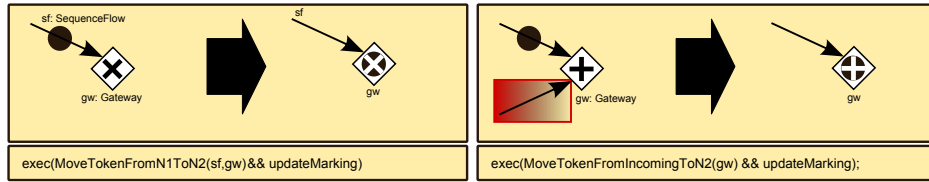


Fig. 4. Visual representation of *enterExclusiveGateway* and *enterParallelGateway* rules

- the path visits the inclusive gateway or
- the path visits a node that has a directed path to a non-empty incoming sequence flow of the inclusive gateway.” [1]

For each empty incoming sequence flow the rule ‘HasNoTokenUpstream’ checks whether that incoming sequence flow has no token upstream (for which the properties do not hold). We refer to the downloadable tool for the definition of this rule.

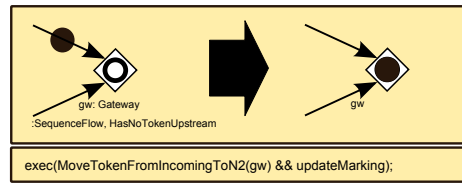


Fig. 5. Visual representation of rule *enterInclusiveGateway*

The sixth rule ‘enterSubprocess’ represents the execution semantics rule for instantiating a subprocess. This rule has two cases, one for the situation in which the subprocess contains a single start event and one for the situation in which the subprocess has no start event. These cases are illustrated in figure 6. In case the subprocess has a single start event, that event get a token when the subprocess is instantiated. In case the subprocess does not have a start event, activities and gateways that do not have incoming sequence flows and that are not the target of a compensation handler get a token when the subprocess is instantiated.

Tokens that are put on elements inside a subprocess instance should be ‘colored’ with that instantiation. This is necessary to prevent that tokens from different subprocess instances are somehow related, for example when determining whether a parallel join gateway is enabled. We leave this for future work.

The seventh rule ‘leaveNodeOneTrans’ represents the execution semantics rule for a node that has a single outgoing sequence flow. It is graphically represented in figure 7. Note that a node in this case also be an intermediate event. We leave the case in which a task has multiple outgoing sequence flows for future work.

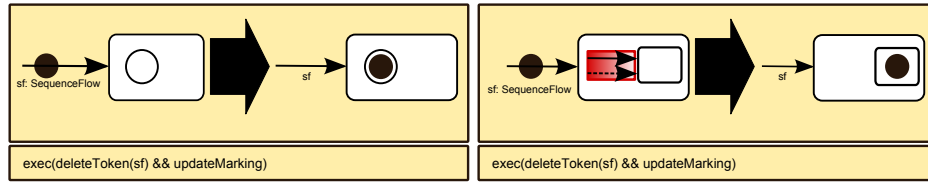


Fig. 6. Visual representation of rule *enterSubprocess*

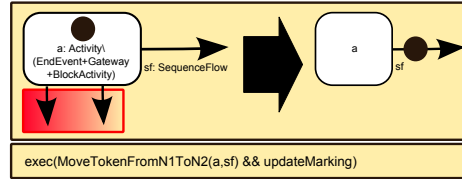


Fig. 7. Visual representation of rule *leaveNodeOneTrans*

The eighth rule ‘leaveSubprocessNormal’ represents the execution semantics rule for completing a subprocess normally. It is graphically represented in figure 7. This rule has two cases, the case in which the subprocess has end events and the case in which the subprocess does not have end events. In the first case the subprocess terminates when there is no token on an activity that is *not* an end event. In the second case the subprocess terminates when there is no token on an activity that has outgoing sequence flows. In both cases the sequence flow that leaves the subprocess receives a token and the tokens that still exist in the subprocess are cleaned up by the ‘cleanupToken’ rule.

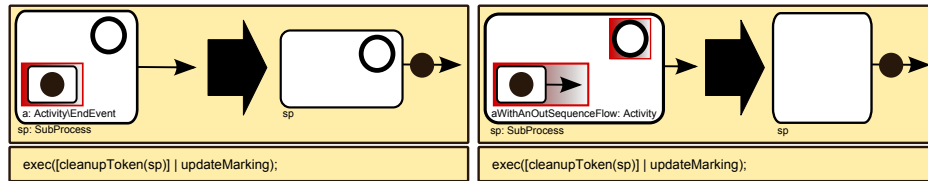


Fig. 8. Visual representation of rule *leaveSubprocessNormal*

The remaining rules apply to leaving gateways. Figure 9 shows the execution semantics rule for an exclusive split gateway. From an exclusive gateway with a token and an outgoing sequence flow, the token is removed from the exclusive gateway and placed on the sequence flow. Therewith, it creates exclusive split behavior, because the token can be put only on a single outgoing sequence flow. Figure 10 shows the execution semantics rule for a parallel split gateway. From a parallel gateway with a token, the token is removed. Subsequently, the rule ‘enableOutflow’ ensures that a token is put on each sequence flow that leaves the

gateway. Figure 11 shows the execution semantics for an inclusive split gateway. This rules for this gateway are similar as those for the parallel split gateway. However, instead of putting tokens on all outgoing sequence flows, a token is put either on the default outgoing sequence flow or on some of the other outgoing sequence flows.

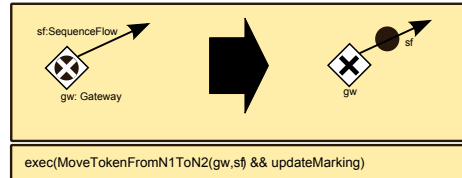


Fig. 9. Visual representation of rule *leaveExclusiveGateway*

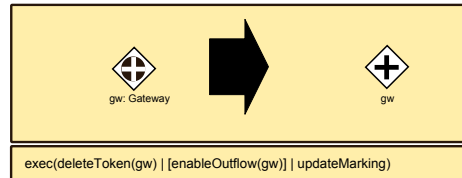


Fig. 10. Visual representation of rule *leaveParallelGateway*

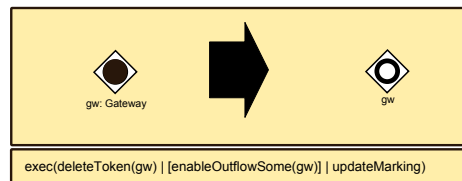


Fig. 11. Visual representation of rule *leaveInclusiveGateway*

3.3 Moving Tokens

The execution semantics rules in the previous subsection use the ‘MoveTokenFromN1ToN2’ and ‘updateMarking’ rules to move tokens and keep track of previous markings.

The rule ‘MoveTokenFromN1ToN2’ is graphically represented in figure 12. This rule moves a token into a place (where a ‘place’ can both be a sequence flow and an activity) and deletes it from another place. Therewith, it effectively moves the token from the place in which it is deleted to the place on which it is

put. The ‘deleteToken’ rule is shown in figure 13. It does not actually delete the token as its name implies, but rather changes the token into a ‘hidden’ token. Hidden tokens are tokens that cannot enable BPMN elements. We use them to keep track of past markings.

The other ‘helper’ rules are defined in a similar manner as ‘MoveTokenFromN1ToN2’ and ‘deleteToken’. These rules are: ‘MoveTokenFromIncomingToN2’, ‘cleanupToken’, ‘enableOutflow’ and ‘enableOutflowSome’.

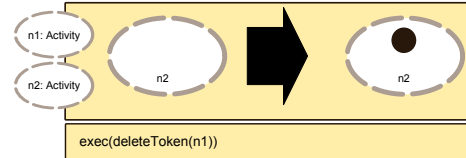


Fig. 12. Visual representation of rule *MoveTokenFromN1ToN2*

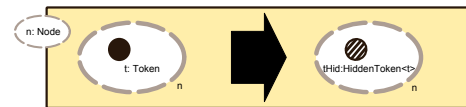


Fig. 13. Visual representation of rule *deleteToken*

The rule ‘UpdateMarking’ is graphically represented in figure 14. This rule finds the last marking in the execution trace. This is a marking that has no pointer ‘Mnext’ to the next marking. It then creates a pointer from that marking to a new marking. Subsequently it executes the rule ‘copyActiveTokenReference’, which creates references from the new marking to all tokens from the previous making that are active (i.e.: not hidden). Also, it executes the rule ‘danglingTokenToMarking’, which creates references from the new marking to all tokens that are not yet referenced by a marking. These are the tokens that are created when firing the last BPMN execution rule. The ‘danglingTokenToMarking’ rule is graphically represented in figure 15.

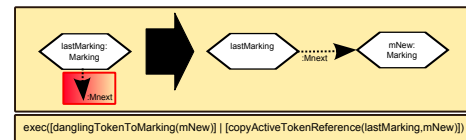


Fig. 14. Visual representation of rule *UpdateMarking*

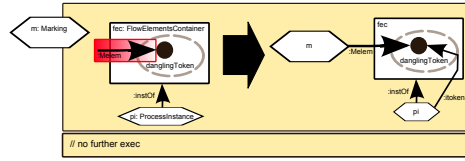


Fig. 15. Visual representation of rule *danglingTokenToMarking*

4 Testing Conformance to Execution Semantics

The goal with which we develop the execution semantics is to use it as a means to check the conformance of tools that implement the BPMN execution semantics, such as workflow engines.

To this end we implemented the execution semantics described in the previous section in a tool called GrGen [5]. This tool can check whether a graph rewrite rule can be executed at a particular time and, if so, it can execute that rule. Since the rewrite rules represent execution semantics rules, this means that we can check whether an execution semantics rule can be executed at a particular time and, if so, we can execute that rule. Using the GrGen implementation of the execution semantics, we can verify conformance of a workflow engine to the BPMN execution semantics as shown in figure 16.

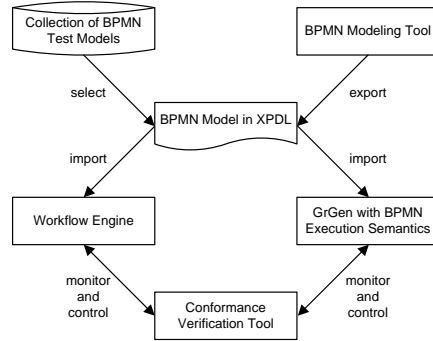


Fig. 16. Conformance Testing of Workflow Engines

The figure shows that we assume XPDL as the interchange format between BPMN modeling tools, workflow engines and the GrGen implementation of the execution semantics. Specifically, we use XPDL version 2.1, because it is the latest stable version of XPDL. The exact version that we use is important, because depending on the version of XPDL that we use, certain BPMN can or cannot be interchanged (in a standard manner). Consequently, the XPDL version that is used determines the BPMN constructs for which we can check the conformance. XPDL version 2.1 supports all sequence flow constructs within the scope of this paper, except for the parallel event-based gateway and the

boundary intermediate event that *does not* cancel the activity to which it is attached (i.e. that has the attribute ‘cancelActivity’ set to ‘false’).

A BPMN model in XPDL can be imported both by a workflow engine (that supports XPDL) and our GrGen implementation of the execution semantics. A verification tool then tests the conformance of the workflow engine. It does this by monitoring and controlling the behavior of both the workflow engine and the execution semantics and verifying that the workflow engine changes state in the same manner as the execution semantics. To this end the verification tool does the following:

1. Determine the execution traces that can be performed by the execution semantics.
2. For each execution trace determine possible values for datafields and in which activity they must be entered.
3. Perform each execution trace in the workflow engine and check whether at each moment in the execution the set of activities that is allowed by the workflow engine is identical to the set of activities that is allowed by the execution semantics.

The implementation of the verification tool is left for future work.

We have also developed a collection of BPMN models to test conformance. Each of the BPMN models in this collection is developed to test a specific execution semantics rule that is defined in the BPMN standard. This collection of models can be used for unified testing, benchmarking and reporting on execution semantics conformance of different workflow engines. A conformance report can indicate specific rules that are or are not correctly implemented by the workflow engine.

5 Related Work

The BPMN 2.0 standard specifies the complete BPMN 2.0 execution semantics in natural language. The use of natural language is sufficiently precise to allow for an intuitive understanding of the execution semantics, but it cannot be directly implemented into a tool for purposes of simulation, verification or execution. Therefore more precise semantics for BPMN have been defined [11, 12, 13, 14, 15, 16, 17]. These semantics differ with respect to the means that is used to specify the semantics, the goal with which the semantics is specified and the conceptual focus of the semantics. Table 1 summarizes them.

Wong and Gibbons [11, 12] define a semantics for a subset of the BPMN control-flow concepts in terms of the process algebra CSP [18]. This semantics allows them to check the consistency of business process models at different levels of abstraction (i.e. refinement checking). It also allows them to specify and check certain properties that must apply to the process. This includes domain specific properties, such as ‘after an order is placed, a response must be sent to the client within 24 hours’, and properties that apply to business process models in general, such as deadlock-freeness and proper completion [19]. We

refer to the latter form of property checking as soundness checking. Dijkman et al. [13] define a semantics for a subset of the BPMN control-flow concepts in terms of classical Petri nets. The goal of their semantics is to define the BPMN execution semantics precisely and to enable soundness checking. Prandi et al. [14] define a semantics in terms of a process algebra called COWS [20]. Their semantics allows for soundness checking of BPMN models and also of quantitative simulation of BPMN models, provided that simulation information is provided with the BPMN model. The semantics is defined for a subset of both the control-flow and the data-flow aspect. Raedts et al. [15] define a semantics for a subset of the BPMN control-flow concepts in terms of classical Petri nets. The goal of their semantics is to enable soundness checking. Dumas et al. [16] define the execution semantics of a particular BPMN construct: the inclusive join gateway. Their goal is to discuss the execution semantics of this particularly complex construct in enough detail to allow animation of BPMN models that use this construct. Takemura [17] defines a semantics for the concepts that are related to BPMN transactions in terms of classical Petri nets. The goal of the semantics is to define the execution semantics of BPMN transactions precisely and to enable soundness checking.

The semantics in this paper differs from the other semantics with respect to the means that are used for the semantics and the prospective use of the semantics. This paper uses graph rewrite rules to define the semantics. One benefit of using graph rewrite rules is that a direct mapping is possible from the execution semantics rules in the BPMN specification to graph rewrite rules. This direct mapping makes the graph rewrite rules easily traceable to BPMN execution semantics rules and easily understandable. Another benefit of using graph rewrite rules is their relative expressive power. For example, classical Petri nets are inherently limited in the semantics that they can represent; it is notoriously hard to represent the OR-join in classical Petri nets and data-related concepts cannot be represented in a feasible manner in classical Petri nets. Such concepts can easily be represented in graph rewriting systems. We aim to exploit this expressive power in future work to define a complete formal semantics of BPMN. These properties enable the use of our semantics for conformance verification. In particular it enables us to compare the execution of a running workflow system to the execution semantics as it is executed in a graph rewriting tool.

6 Conclusions

This paper presents a formalization of a subset of the BPMN 2.0 execution semantics in terms of graph rewrite rules. The paper shows that there is a direct correspondence between the rules that informally define the BPMN 2.0 execution semantics in the standard and the graph transformation rules that formally define the BPMN 2.0 execution semantics in this paper. The benefit of having this direct correspondence is that the formalization can easily be validated, because the formal definition of each rule can be checked with its informal specification in the BPMN 2.0 standard and the completeness of the rules can be determined

Table 1. Semantics defined for BPMN

Semantics	Means	Uses	Focus
BPMN Standard [1]	Natural language	Semantics specification	Complete
Wong and Gibbons [11, 12]	CSP	Refinement checking Property checking Soundness checking	Control-flow subset
Dijkman et al. [13]	Petri nets	Semantics specification Soundness checking	Control-flow subset
Prandi et al. [14]	COWS	Soundness checking Quantitative simulation	Control-flow subset Data-flow subset
Raedts et al. [15]	Petri nets	Soundness checking	Control-flow subset
Dumas et al. [16]	Pseudo code	Semantics specification	OR-Join
Takemura [17]	Petri nets	Semantics specification Soundness checking	Transactions
This paper	Graph rewriting	Semantics specification Conformance checking	Control-flow subset

by checking whether the complete specification of the execution semantics is covered.

The formalization of the execution semantics in this paper can be used, among other things, for simulation, animation and execution of BPMN models. For this purpose it was implemented into a tool called GrGen. In particular we aim to use the formalization as a tool to verify the conformance of workflow engines that execute BPMN models to the BPMN execution semantics. To this end we will implement a monitoring and conformance verification tool in future work.

Another purpose for which formal semantics of process models are often used is for verification of correctness of those process models. The semantics that we defined in this paper is less suitable for this purpose, because it makes no claims about the statespace that can be constructed with it, such as whether the statespace is finite or whether the algorithm to verify the reachability of certain states can complete. However, formal semantics of BPMN in terms of Petri-nets or Process Algebra exist and can be used for that purpose.

It is the aim of this work to formalize the execution semantics of BPMN in its entirety, also covering the data and resource aspects of BPMN. To that end we will implement formal execution semantics rules for the data and resource aspect in future work. We will also complete the formal execution semantics for the control-flow aspect.

References

1. Object Management Group: Business process model and notation beta 1 for version 2.0. Technical Report dtc/2009-08-14, Object Management Group, Needham, MA, USA (2009)

2. Workflow Management Coalition: Process definition interface – XML process definition language version 2.1a. Technical Report WFMC-TC-1025, Workflow Management Coalition, Hingham, MA, USA (2008)
3. Workflow Management Coalition: XPD L implementations (June 2010) Accessed 21 May 2010 from: <http://www.wfmc.org/xpd1-implementations.html>.
4. Rozenberg, G., ed.: Handbook of Graph Grammars and Computing by Graph Transformation. Volume I: Foundations. World Scientific Publishing Co., Inc., River Edge, NJ, USA (1997)
5. Jakumeit, E., Buchwald, S., Kroll, M.: GrGen.NET. International Journal on Software Tools for Technology Transfer (STTT) (2010)
6. Heckel, R.: Tutorial introduction to graph transformation. In: ICGT '08: Proceedings of the 4th international conference on Graph Transformations, Berlin, Heidelberg, Springer-Verlag (2008) 458–459
7. Habel, A., Pennemann, K.h.: Correctness of high-level transformation systems relative to nested conditions. Mathematical Structures in Comp. Sci. **19**(2) (2009) 245–296
8. Van Gorp, P., Mazanek, S., Rensink, A.: Transformation Tool Contest – Awards. <http://is.ieis.tue.nl/staff/pvgorp/events/TTC2010/?page=Awards> (2010)
9. Van Gorp, P.: BPMN semantics: online virtual machine. <http://is.ieis.tue.nl/staff/pvgorp/share/?page=ConfigureNewSession&vdiID=364> (2010)
10. Budinsky, F., Brodsky, S.A., Merks, E.: Eclipse Modeling Framework. Pearson Education (2003)
11. Wong, P.Y., Gibbons, J.: A process semantics for BPMN. In: Proceedings of the 10th International Conference on Formal Methods and Software Engineering. Volume 5256 of Lecture Notes In Computer Science. (2008) 355–374
12. Wong, P.Y., Gibbons, J.: Formalisations and applications of BPMN. Science of Computer Programming **In Press, Corrected Proof** (2009)
13. Dijkman, R., Dumas, M., Ouyang, C.: Semantics and analysis of business process models in bpmn. Information and Software Technology (IST) **50**(12) (2008) 1281–1294
14. Prandi, D., Quaglia, P., Zannone, N.: Formal analysis of BPMN via a translation into COWS. In: Proceedings of COORDINATION 2008. Volume 5052 of Lecture Notes in Computer Science. (2008) 249–263
15. Raedts, I., Petkovic, M., Usenko, Y., van der Werf, J., Groote, J., Somers, L.: Transformation of BPMN models for behaviour analysis. In: Proceedings of the 5th International Workshop on Modelling, Simulation, Verification and Validation of Enterprise Information Systems, INSTICC Press (2007) 126–137
16. Dumas, M., Grosskopf, A., Hettel, T., Wynn, M.: Semantics of standard process models with or-joins. In: Proceedings of OTM 2007, Part I. Volume 4803 of Lecture Notes in Computer Science. (2007) 41–58
17. Takemura, T.: Formal semantics and verification of BPMN transaction and compensation. In: Proceedings of the IEEE Asia-Pacific Conference on Services Computing, Los Alamitos, CA, USA, IEEE Computer Society (2008) 284–290
18. Roscoe, A.W.: The Theory and Practice of Concurrency. Prentice-Hall (1998)
19. van der Aalst, W.: Verification of workflow nets. In: Proceedings of the 18th International Conference on Application and Theory of Petri Nets. (1997) 407–426
20. Prandi, D., Quaglia, P.: Stochastic COWS. In: Proceedings of ICSOC 2007. Volume 4749 of Lecture Notes in Computer Science. (2007) 245–256