Information and Software Technology 55 (2013) 365–394

Contents lists available at SciVerse ScienceDirect

Information and Software Technology

journal homepage: www.elsevier.com/locate/infsof

A visual token-based formalization of BPMN 2.0 based on in-place transformations

Pieter Van Gorp *, Remco Dijkman

Eindhoven University of Technology, PO Box 513, 5600 MB Eindhoven, The Netherlands

ARTICLE INFO

Article history:
Received 20 December 2011
Received in revised form 29 August 2012
Accepted 31 August 2012
Available online 15 September 2012

Keywords:
BPMN
BPM
MDA
Formal semantics
Graph transformation

ABSTRACT

Context: The Business Process Model and Notation (BPMN) standard informally defines a precise execution semantics. It defines how process instances should be updated in a model during execution. Existing formalizations of the standard are incomplete and rely on mappings to other languages.
Objective: This paper provides a BPMN 2.0 semantics formalization that is more complete and intuitive than existing formalizations.
Method: The formalization consists of in-place graph transformation rules that are documented visually using BPMN syntax. In-place transformations update models directly and do not require mappings to other languages. We have used a mature tool and test-suite to develop a reference implementation of all rules.
Results: Our formalization is a promising complement to the standard, in particular because all rules have been extensively verified and because conceptual validation is facilitated (the informal semantics also describes in-place updates).
Conclusion: Since our formalization has already revealed problems with the standard and since the BPMN is still evolving, the maintainers of the standard can benefit from our results. Moreover, tool vendors can use our formalization and reference implementation for verifying conformance to the standard.

1. Introduction

The Business Process Model and Notation (BPMN) version 2.0 [41] is a standard notation for business process modeling. It presents a set of concepts and notational elements for business process modeling. It also presents an execution semantics that defines precisely how models in the BPMN notation should behave when executed in a tool. That semantics is defined informally using natural language.

There exist various initiatives to define a formal execution semantics in addition to the informal one [62,63,10,45,46,12,56]. These formal semantics are defined for a wide variety of reasons, including: enabling formal reasoning about the correctness of BPMN 2.0 process models, enabling simulation of those models and reasoning about the correctness of the standard. All of these formalizations rely however on a mapping (i.e., an out-of-place transformation [35]) of BPMN elements to elements in another formalism (such as Petri nets). This paper presents a formalization using visual, in-place [35], transformation rules. Defining the execution semantics using in-place transformation rules has three important benefits.

Fistly, the approach is intuitive since graph transformation rules can be defined by using the BPMN 2.0 notation itself.

Secondly, the approach is simple since there is good traceability between the informal execution semantics rules in the standard and their formal counterparts in a graph transformation form. This facilitates easy validation of the correctness of each of the formal rules. The traceability exists because the informal semantics in the standard is also defined in terms of a token-game. It implicitly relies on rules that specify when a certain notational element can be (de-)activated and what happens when it does. This can be mapped easily to a graph transformation rule, which always have a "match" part and a "rewrite" part. We have designed our rule set such that each notational element has two rules: one for activating the element and one for disabling the element. Regardless of the rule representation syntax (formal or informal, textual or visual), this consistent design already reduces complexity since the semantics of different elements can be considered in isolation.

Finally, using graph transformation rules allows the formalization to be complete. Theoretically, it is possible to develop a complete execution semantics of BPMN 2.0 in terms of graph transformation rules, because graph transformation is Turing complete [21]. This as opposed to, for example, classical Petri nets, in terms of which some constructs are notoriously hard to represent [10]. As a proof of this concept, our execution semantics covers more rules from the BPMN 2.0 standard than any other formal semantics so far.

* Corresponding author. Tel.: +31 40 2472062; fax: +31 40 2432612.
E-mail addresses: p.m.e.v.gorp@tue.nl (P. Van Gorp), r.m.dijkman@tue.nl (R. Dijkman).

0950-5849/$ - see front matter © 2012 Elsevier B.V. All rights reserved.
http://dx.doi.org/10.1016/j.infsof.2012.08.014

There exists a wide variety of graph transformation tools that can execute graph transformation rules. Therefore, our execution semantics in terms of graph transformation rules can be verified on a test-suite of complex BPMN models. This has already enabled us to verify the expected behavior of our formalization. Moreover, we have discovered various points for improvement for the BPMN standard and this paper enables others to extend this work.

We therefore recommend the maintainers of the standard to (1) validate whether our formalization matches their intentions, (2) to use our supportive prototype as an instrument to improve the informal text in the standard and (3) to include an appendix based on our visual, rule-based formalization. This should lead to a better conformance to the standard and a better adoption of language constructs (especially non-trivial ones such as compensation events).

The remainder of this paper is structured as follows. Section 2 provides an introduction to graph transformation and BPMN 2.0. Section 3 defines the BPMN 2.0 execution semantics formally using graph transformation rules. Section 4 explains how we have constructed a reference implementation, how that implementation can be used and which alternatives we consider promising. Finally, Section 5 presents the evaluation of our contribution, Section 6 presents related work and Section 7 concludes.

2. Preliminaries

Before elaborating on the challenging topic of BPMN 2.0 semantics definition, Section 2.1 provides a gentle introduction to the BPMN. Section 2.2 then demonstrates our rule-based approach to semantics formalization on a language with a notoriously more simple semantics than the BPMN. That section serves as a "hello world" teaser to the next sections, which are more technically detailed. Section 2.3 provides a systematic introduction to typed attributed graphs and graph transformation. Finally, Section 2.4 introduces techniques for composing primitive rules into more powerful units.

2.1. BPMN 2.0

BPMN 2.0 can be used to create models of an organization's business processes. To this end, it defines a large number of notational elements, the meaning of those elements and an execution semantics that defines how certain combinations of elements should behave.

Fig. 1 shows a simple BPMN model of an order handling process. The model starts with a start event, represented by a circle, that is triggered when a message, represented by the envelope icon, arrives. The message contains an order. After the order arrives, the organization starts to process the order in a subprocess, represented by a rounded rectangle that contains other elements. The subprocess contains two activities, represented by rounded rectangles, and can be interrupted when a cancelation (represented by the cross symbol attached to the subprocess) about the order is received. After either the subprocess completes or an order cancelation is received, the alternative paths are joined by a so-called exclusive gateway, represented by the diamond with the "X". Finally, the process reaches an end event, representing completion.

The elements shown in Fig. 1 can be interpreted as a graph: the start event, end event, subprocess element, its contained activities, its attached message event and finally the exclusive gateway are the *nodes* of the graph while the flow arcs shown in Fig. 1 are its edges. Finally, for representing the hierarchical relation between the subprocess node and its children we can also assume the presence of a specificly typed edge between these elements.

The execution semantics of BPMN 2.0 is defined in terms of a large number of execution semantics rules. One of these rules, for example, states that the behavior of an exclusive gateway is such that: *"Each token arriving at any incoming Sequence Flows activates the gateway"*. We will introduce the graph-based formalization of this rule very gently in Section 2.3 but already demonstrate its expected behavior by means of Fig. 2. Note that the BPMN standard provides no standard icon for representing tokens, which makes it impossible to visualize process executions in standard syntax. We represent tokens as black dots, inspired by other flow-based languages such as classical Petri nets. Also note that Fig. 2 shows one process state per numbered item (1–14). Such a state consists of all process elements, all tokens, and all process instances to which these tokens belong. Again, inspired by Petri nets, we refer to these states as *Markings* and to the overall graph in Fig. 2 as a *statespace*.

Fig. 2 shows all possible executions of the order handling process (cfr., Fig. 1). The rule for executing exclusive gateways is triggered for transitioning from state 6 to 7 as well as from state 11 to 12. Clearly, for these transitions, the aforementioned behavior is satisfied. Note that the formal semantics from this paper enables the execution of BPMN models even when no guard expressions have been specified by the business analyst: all choices can be made non-deterministically or based on additional user input during process execution. We argue that this is quite valuable especially for business analysts, who are typically not trained in guard expression specification languages. For example when designing the process models from Fig. 1, the analyst may find it useful to double-check that regardless of arc inscription details there are two ways to complete the process and that in both final states (markings 9 and 14 from Fig. 2) there is exactly one token in the end event.

The figure also illustrates that upon order cancelation, tokens are removed from the subprocess elements as well as from the subprocess activity (cfr., the three transitions leading to state 10). Finally, note that although they are identical visually, state 7 is different from state 12 (and 8 from 13 and 9 from 14 respectively). More specifically, the state of the subprocess is "completed" for

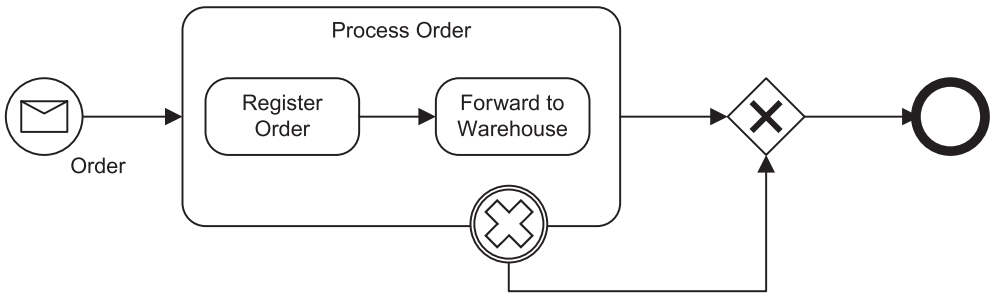


Fig. 1. Example BPMN 2.0 model: processing orders.

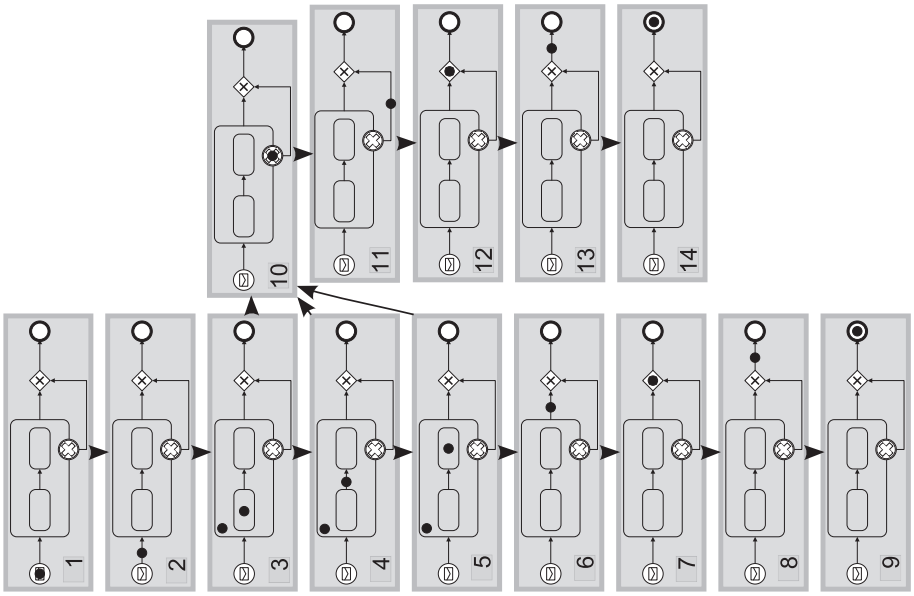


Fig. 2. Statespace showing all conceptually possible execution paths of the example process.

states 6–9 while it is "canceled" for states 10–14. These node differences could be visualized using a proprietary notation but for the sake of conciseness we hide process instance nodes from Fig. 2.

The remaining BPMN 2.0 notational elements and execution semantics rules will be gradually introduced in Section 3. For precision, Appendix A presents an abstract syntax of the BPMN 2.0 notational elements in terms of mathematical sets. That appendix also clarifies the relation to specific storage formats. Appendix B presents an example, in which the definition from Appendix A is used to formalize the example from Fig. 1.

2.2. Petri net semantics formalized as graph transformation rules

This section introduces the reader to the use of graph transformation rules for the semantics definition of visual modeling languages. More specifically, we clarify how the semantics of classical Petri nets (cfr., [39]) can be defined very easily. We intentionally post-pone the systematic introduction to graph transformation (cfr., Section 2.3) since this example is largely self-explaining. This clarifies that the complexity of subsequent BPMN oriented rules is mostly due to the complexity of the BPMN constructs under consideration and not due to the complexity of our approach.

Fig. 3 shows the source code listing of the graph transformation rule implementation in GrGen.NET[1] syntax. Line 1 opens the definition of a rule named *step*. Line 2 declares a node *t* in the left-hand side of the rule. This node represents a transition that satisfies the condition for *firing* [39]. Lines 3–8 express that firing condition formally: there should be no *Place* before *t* that does not contain a token. Put differently, all places before *t* should have at least one token. Lines 9–14 and 15–20 show two subrules that respectively (1) remove one token from each input place and (2) add one token to each output place. Remark that the right-hand side of the compound rule (shown on line 21) contains no side-effects. Therefore, the *t* node is preserved without changes.

Although the GrGen.NET code from Fig. 3 contains already much less technicalities (e.g., iterators) than a corresponding

```
1 rule step {
2   t:Transition;
3   negative {
4     pPreNAC:Place -:pre-> t;
5     negative {
6       pPreNAC -:tokens-> . ;
7     }
8   }
9   iterated {
10    tok:Token <-:tokens- pPre:
          Place -:pre-> t;

11    modify {
12      delete(tok);
13    }
14  }
15  iterated {
16    t -:post-> pPost:Place;
17    modify {
18      pPost -:tokens-> :Token;
19    }
20  }
21  modify {}
22 }
```

Fig. 3. The Petri net operational semantics defined in GrGen.NET syntax.

implementation in an imperative programming language [60], we propose to document it visually. We make transformation rules more easily understandable by visualizing them in the notation of the target language. Fig. 4 shows the visual counter-part of Fig. 3, in Petri net syntax. Note that the algebraic operator "&&" is used to compose subrules. Also note that the *NAC* blocks correspond to *negative* blocks in GrGen.NET code. The nested NAC blocks from Fig. 4 express that: (1) there should be no input place, for which (2) there is no token on it.

Fig. 5 illustrates the tool support for debugging transformation rules. The figure shows a screenshot of an example execution sequence of our *step* rule. The top of the figure shows a graph visualizer whereas the middle of the figure shows an interactive rule execution shell. The example is based on an input Petri net with 11 places and 7 transitions. The graph visualizer has been configured such that (1) Transition nodes are displayed as gray

[1] We use GrGen.NET [18,30] since it represents the state-of-the-art in transformation tools [61]. An informal and formal platform description is available in [5] and [17] respectively.

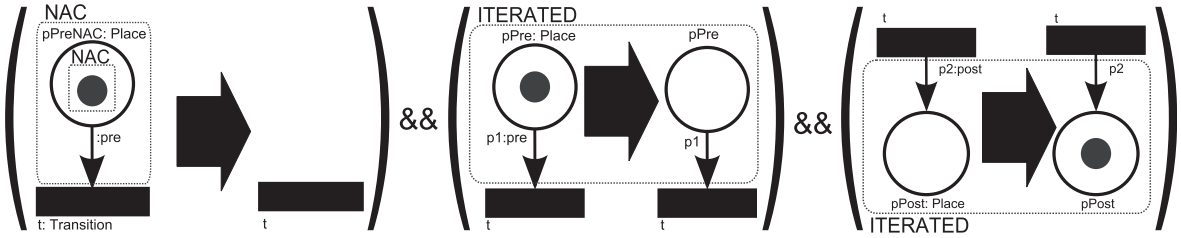


Fig. 4. Visual representation of the graph transformation rule defining for Petri nets.

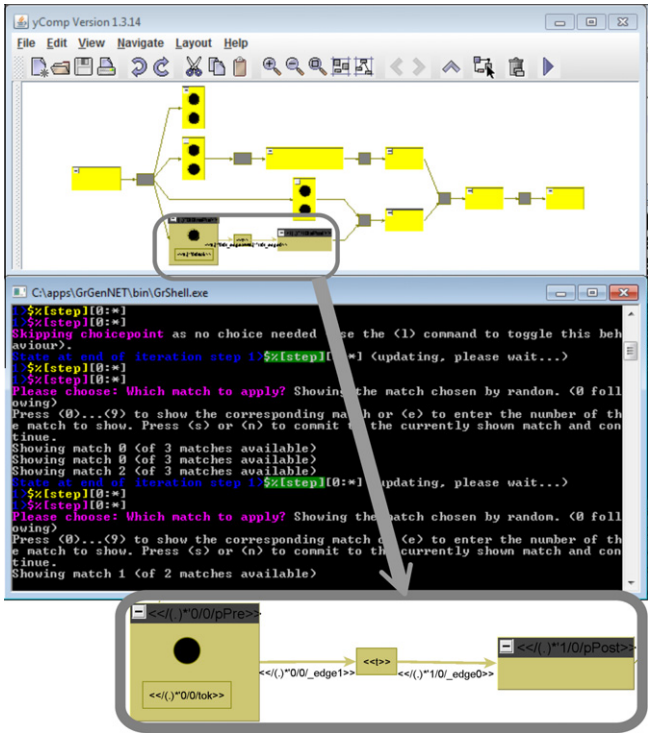


Fig. 5. Tool support for executing and debugging graph transformation rules.

rectangles, (2) Place nodes are displayed as yellow rectangles, and (3) Token nodes are displayed as black bullets that are visually embedded in their corresponding places. On the snapshot from Fig. 5, the *step* rule has been executed twice already. These rule applications have moved two tokens from the place at the very left of the diagram to the subsequent four places. The interactive shell now indicates that the *step* rule matches on two occurrences in the graph.

The first match is currently highlighted by the graph visualizer and further clarified by the magnification of the three nodes that constitute the match. The enlarged fragment shown at the bottom of Fig. 5 shows that the graph visualizer even shows the names of the transformation rule variables of the *step* rule: the node at the left of the fragment matches the ≪*pPre*≫ variable from the rule definition (cfr., line 10 from Fig. 3 as well as the subrule in the middle of Fig. 4). The node at the left of the enlarged fragment of Fig. 5 contains two other nodes: the black bullet at the top represents a token that is *not* matched by this occurrence of the *step* rule whereas the box labeled as ≪*tok*≫ is matched by the variable defined on line 10 of Fig. 3. Similarly, ≪*t*≫ matches the variable from line 2 and ≪*pPost*≫ matches the variable from line 16.

The other match of the *step* rule could be enabled by pressing the appropriate keys in the shell shown at the bottom of Fig. 5. Other debugging functionalities are available but most importantly the figure illustrates that advanced tool support is available for putting the theoretical graph transformation based formalization of the Petri net semantics into practice. This is particularly useful while developing the formal semantics since errors are easily made throughout that process. This motivates why we considered this approach promising for the formalization of complex BPMN 2.0 constructs. Even readers that have no background in graph transformation should now be convinced of that potential too and they can use the following two sections as a more systematic introduction to the fundamentals of that technology.

2.3. Typed attributed graphs and graph transformation

The brief tutorial from Section 2.2 applies the paradigm of attributed graph transformation with node type inheritance [8]. This paradigm is based on theoretical foundations that emerged in the 1970s [15]. Therefore, we can rely safely on high level abstractions without introducing ambiguity. Others are leveraging the theoretical foundations to build analysis tools [34]. In the context of this paper, we provide intuitive explanations of the concepts of attributed graph, typed attributed graph and graph transformation rule. We refer to [8,15,34] for more precise and formal definitions.

An *attributed graph* is a graph in which both nodes and edges can have attributes. Attributes are themselves nodes, but are distinguished from regular nodes. As attributed graphs, we use the concept of E-graphs [13]. E-graphs however also support edge attributes, which are not used by this paper and hence omitted for the sake of simplicity (inspired by Heckel et al. [25]). In a *typed, attributed graph* (TAG) a set of node types and a set of edge types exist and each node and each edge is mapped to a node type or edge type, respectively. A *type graph* is a distinguished graph where nodes and edges are considered as node and edge types, respectively [4]. Fig. A.57 in Appendix A illustrates how the node and edge types can be encoded in a type graph.

A *graph transformation rule* defines how one TAG can be rewritten into another TAG. It consist of two TAGs which are called the left-hand side (LHS) and right-hand side (RHS). The LHS and RHS graph of a rule need to be compatible in the sense that their intersection should be a graph again. For the application of a graph transformation rule to a host graph the following simplified algorithm can be used:

1. Identify the LHS graph in the host graph. This involves finding a total graph morphism[2] that matches the left-hand side in the host graph. Unless specified otherwise, the morphism should be *isomorphic*, which means that each element from the LHS graph should be mapped to at most one element in the host graph (i.e., the mapping should be *injective*).

[2] This morphism should take into account subtyping, as formalized by de Lara et al. [8].

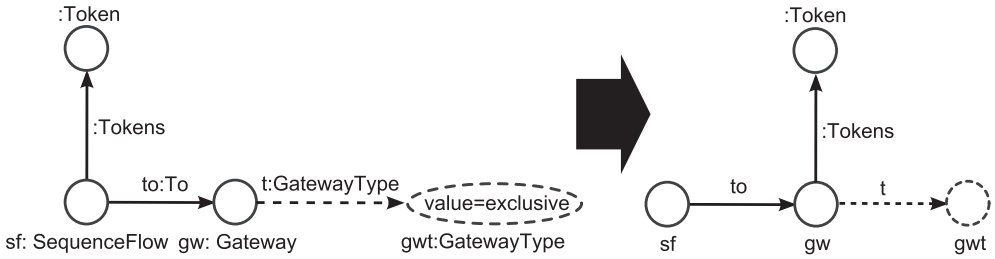


Fig. 6. Graphical representation of a graph transformation rule (flat graph syntax).

2. Delete all corresponding graph elements from the host graph, that are part of the LHS graph but not part of the RHS graph. Node deletions may lead to dangling edges. In our approach, such edges are deleted implicitly.
3. Create a corresponding element in the host graph for each element in the RHS graph that is not in the LHS graph.
4. Evaluate attribute updates that are defined on elements in the graph transformation rule to their corresponding elements in the host graph.

Using this definition, graph transformation rules can easily be graphically represented, by drawing the left and right-hand side graphs of the rule, including the types of the nodes and edges involved. To be able to identify nodes and edges that are the same in the left and right-hand side of the rule, nodes and edges can be given identifiers in the context of the rule. For example, Fig. 6 illustrates a rule that searches for a pattern of three graph nodes with two graph edges between them. One of these graph nodes has an attribute edge *t* to an attribute node *gwt*. (Attribute elements are graphically represented with a dashed line.) Attribute node *gwt* from the LHS graph should be mapped to an attribute node of type *GatewayType* with value exclusive.

The rule rewrites each occurrence of this pattern by deleting the edge of type *Tokens* and its attached node of type *Token*, because these elements appear in the left-hand side but not in the right-hand side. The rule adds another node of type *Token* as well as a new edge of type *Tokens*, because these elements appear in the right-hand side, but not in the left-hand side. Remark that the new edge originates from *gw* instead of from *sf*. For simplicity, attribute nodes and attribute edges are graphically embedded in their graph nodes in the remainder of this paper. Fig. 7 follows this style to represent the same rule as that from Fig. 6.

This paper relies on some syntactical short-hands to make the left- and right-hand sides of our transformation rules resemble even more the standard BPMN 2 syntax. Fig. 8 illustrates the three most important short-hands:

1. We pretty-print patterns with specific properties using a special icon. For example, nodes of type *Gateway* and an associated *GatewayType* attribute of value *exclusive* are shown with the expected icon of a BPMN exclusive gateway (i.e., a diamand with an embedded "X").
2. We pretty-print a sequence flow node and its *From* and *To* edges as an *edge* (with obvious source and target values).
3. We abstract from edges of type *Tokens*: we do not show these edges explicitly but draw the token elements on the originator of the edge instead.

Fig. 8 shows the pretty-printed counter-part of the rule shown in Fig. 7. The figure illustrates that the pretty-printed syntax is on the one hand less verbose than a conventional graph representation (due to the other representation of sequence flows) and on the other hand more recognizable to BPMN experts (due to BPMN-specific node graphics). Also note that arcs with a solid line look exactly like a BPMN sequence flow arc. Therefore, we agree that such arcs are implicitly of type *Sequence Flow* for all BPMN rules in this paper. Arcs of other types are drawn with dashed lines in all rule representations to clearly distinguish them from sequence flows.

Fig. 7. Graphical representation of a graph transformation rule (attributed graph syntax).

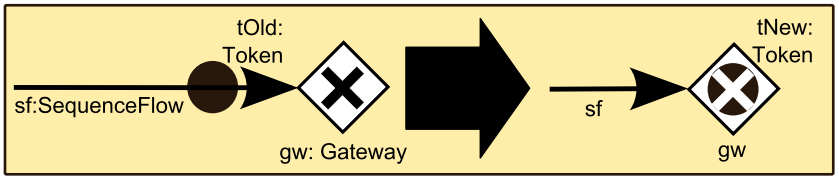


Fig. 8. Graphical representation of a graph transformation rule (pretty-printed syntax).

For a more comprehensive formal treatment of rule applications in terms of Category Theory, we refer to the handbook of graph grammars [14]. This paper adopts the so-called single pushout (SPO [18]) approach, which means that after rule application, all dangling edges (if any) are removed automatically. There exist various extensions to the basic mechanisms of graph transformation and innovations in this area are the subject of ongoing research [55]. Most practical graph transformation languages include a set of control flow constructs [53,24] that enable the use of variables and logical as well as iterative control flow. The next section introduces the key extensions that are applied in this paper.

2.4. Algebraic transformation operators

There exist algebraic operators that increase the expressive power of graph transformation rules and/or and that also make these rules more easy to maintain. In this section we explain these operators in an intuitive manner, insofar we use them in Section 3 to define the BPMN 2.0 execution semantics. Fig. 9 shows the extensions, both in their graphical and algebraic form.

Three additional operators can be used to enrich pattern specifications in the left-hand side of a rule. We allow for the use of the && operator to represent that both graphs must be found for the pattern to match and the || operator to represent that one of the graphs must be found for the pattern to match. These operators evaluate 'lazily', meaning that H is only evaluated if that is strictly necessary (i.e., only if G matches in the case of G && H; and only if G does not match in the case of $G \parallel H$). Note that both these operators are shorthands that can be rewritten. G && H can be rewritten as a single graph and $G \parallel H \rightarrow I$ can be rewritten as $G \rightarrow I \parallel H \rightarrow I$, as it will be defined later. The ! operator represents the negative application condition (NAC [22]). NACs enable the specification of a pattern that should *not* be matched when checking the applicability of a rule [32].

Rules can be composed as a whole using algebraic operators in the same notation [27]. If r and s are rules, $r \parallel s$ is also a rule that applies r or s ('lazily', such that s can only be applied if r cannot) and succeeds if either r or s can be applied. If r and s are rules, r && s is a rule that applies r and s ('lazily', such that s can only be applied if r can be applied) and succeeds if both r and s can be applied. If r is a rule, $r*$ is also a rule that is applied at once on as many occurrences as possible. $r*$ always succeeds. We also apply a variant of $r*$ that accepts a lower and upper bound (l and u) as parameters. $r*[l...u]$ applies r at once on a random number of occurrences between l and u. It fails if less than l occurrences are available. If r is a rule, $r?$ is also a rule that is applied if it can be applied. $r?$ always succeeds. If $G \rightarrow I$ is a rule, that rule can apply another rule r, in addition to rewriting G into I, if a match for G can be found. This is denoted as $G \rightarrow I$ && r. Since r is in the right-hand side of the compound rule, it does not affect the overall match (i.e., the compound rule succeeds even if r cannot be applied). Moreover, unless explicitly stated otherwise, its variables are allowed to be mapped non-injectively with context variables from G.

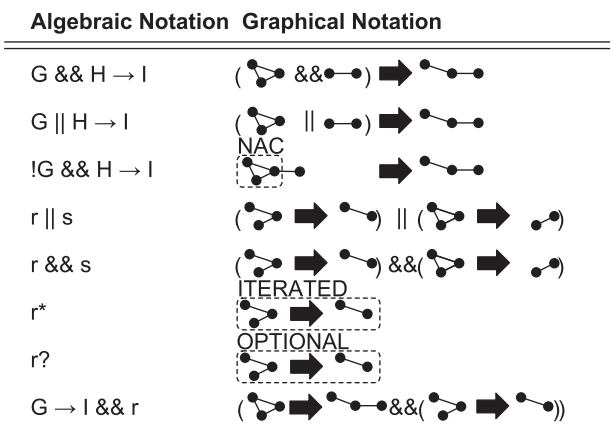


Fig. 9. Algebraic operators and shorthands (see [29] for details).

Note that it is possible to create a rule of the form $G \rightarrow G$ that does not modify the graph (as the right-hand side of the rule is identical to the left-hand side) but that can be used to test whether the graph has a certain structure. We will also simply write G when using such rules. Also note that it is possible to give rules a name and then (recursively) be invoked by other rules, using this name.

Finally, we point out that this section builds upon formalizations that have only been published in German so far. Therefore, we call the theoretical graph transformation community to arms: there is an urgent need to strengthen the international body of knowledge with a comprehensive formal treatment of the rule composition concepts presented here. While this section can serve as an overview, the German Ph.D. and master theses of respectively Rubino Geiß [17] and Edgar Jakumeit [28] provide a promising basis regarding category and star pair graph grammars. Formalizations of other languages with advanced nesting or control structures may also support the formalization of the concepts employed here [31,48,23,43].

3. Execution semantics

This section defines the execution semantics of BPMN 2.0 in terms of graph transformation rules. The section has the same structure as the BPMN 2.0 specification's [41] chapter on the execution semantics, to maintain good traceability between the execution semantics rules in the specification and the graph transformation rules that formalize them.

Fig. 10 provides an overview of the BPMN 2.0 concepts for which we define the execution semantics. The tables show the execution semantics rule in the BPMN 2.0 standard that is formalized, the graphical notation of the concept that is formalized and the names of the graph transformation rules that realize the formalization. For the special event types (message, error, compensation and signal), only the specialized rules are listed. Other than for these rules, the events behave as typical start, intermediate or end events.

For the sake of conciseness, the text of this section focuses on those rules that advance the state of the art in BPMN formalization. Rules that are relatively simple are therefore discussed just briefly. We do include their visual representation to illustrate the breadth and depth or our contribution.

3.1. Process instantiation and termination

3.1.1. Autonomous instantiation

Fig. 11 shows the instantiation rule for top-level processes. The rule specifies that such a process can be instantiated and started autonomously for two types of start events (normal ones and timed ones): according to the attribute constraints (e.g., [*this.TriggerType==None*]), the *this* node of type *StartEvent* in the left-hand side should either have its *TriggerType* attribute set to *None* or to *Timer* (cfr., $\mathcal{D}_{trigtype}$ in Appendix A). Since these attribute constraints are following from the BPMN symbols for the two *this* nodes in Fig. 11, they can also be omitted from the figure. For the sake of conciseness, that style is adopted for subsequent rules.

Note that by applying OR composition of two patterns in the rule's left-hand side, we do not have to maintain two separate rules with identical right-hand sides. This syntactical convenience does not affect the formal background of the underlying rewriting

Activity Type	Notation	Rewrite Rule
Process Instantiation / Termination		enterAutonomousStartEvent completeProcessNormal leaveProcessNormal
Sequence Flow Considerations		leaveTaskOneOut
		leaveTaskMoreOut
		leaveImplicitInclusiveOut catchImplicitlyThrownException
		enterTask
Sub processes / Call activity		enterSubprocess reEnterSubprocess completeProcessNormal leaveSubprocessNormal
Loop activity		reEnterLoopActivity reEnterLoopSubprocess skipLoopActivity
Parallel gateway		enterParallel leaveParallel
Exclusive gateway		enterExclusive leaveExclusive catchImplicitlyThrownException
Inclusive gateway		enterInclusive leaveInclusive catchImplicitlyThrownException

Event Type	Notation	Rewrite Rule
None Start events		enterAutonomousStartEvent, enterSubProcess, leaveStartEvent
Intermediate events		enterIntermediateThrowEvent (See Compensation, Message, Signal)
Intermediate boundary events		enterAutonomousBoundaryEvent, leaveAutonomousBoundaryEvent, catchImplicitlyThrownException
None End events		enterEndEvent, completeProcessNormal
Terminate End events		enterEndEvent, leaveTerminateEndEvent
Message events		enterIntermediateThrowEvent, enterEndEvent, leaveMessageThrowEvent, enterMessageCatchIntermediateEvent, enterMessageCatchStartEvent
Error events		enterThrowErrorEvent, leaveThrowErrorEvent
Compensation events		enterCompensationEndEvent, enterCompensationIntermediateThrowEvent, leaveCompensationIntermediateThrowEvent, UndoProcessInstance, completeSubprocessCompensation
Signal events		enterIntermediateThrowEvent, enterEndEvent, leaveSignalThrowEvent, enterSignalCatchIntermediateEvent, enterSignalCatchStartEvent

Fig. 10. Overview of BPMN 2.0 concepts with execution semantics rules.

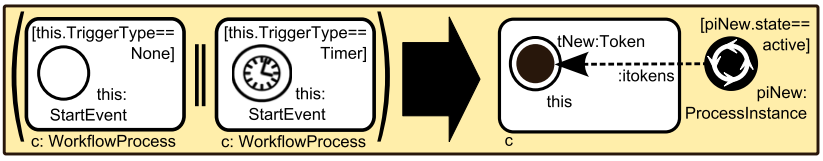


Fig. 11. Autonomous process instantiation: rule *enterAutonomousStartEvent*.

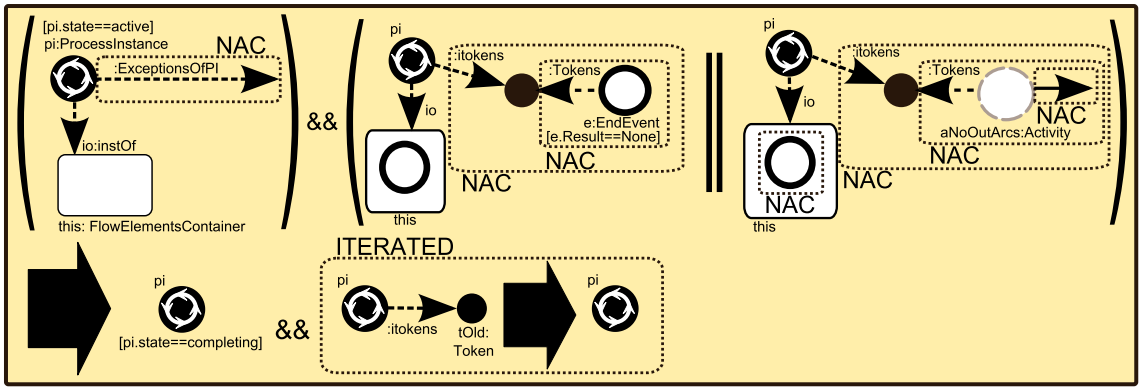


Fig. 12. Normal termination of a process: rule *completeProcessNormal*.

paradigm but it is of great practical benefit. Note also that for readability purposes we have introduced a new icon for representing *ProcessInstance* nodes (cfr., *piNew* in Fig. 11).

The right-hand side in Fig. 11 formalizes that upon process instantiation, a token is added to the start event of that process. Additionally, a new *ProcessInstance* is created and its state is set to *active*. The newly created token is associated with the new process instance via an edge of type *itokens*.

3.1.2. Normal termination

Fig. 12 shows rule *completeProcessNormal*, which ensures that a subprocess (or top-level process) is completed at the right moment. This moment is specified by the rule's two AND-composed subpatterns in its left-hand side. The first subpattern (at the left of the && operator) states that the rule applies to situations where a process instance is in the *active* state. The NAC in this subpattern states that no exceptions should have been thrown for this process

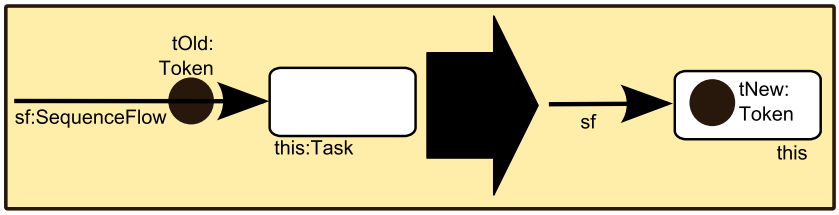


Fig. 13. Regular sequence flow for task elements (1/2): rule *enterTask*.

instance. The second subpattern (i.e., the subpattern after &&) states that additionally at least one of the following conditions needs to hold: (a) the process holds an end event and the process does not hold a token that is not in an ordinary end event, or (b) the process has no end events and no tokens that are not in an activity without outgoing arcs. If this second subpattern matches too, the complete left-hand side matches and the right-hand side can be applied.

The right-hand side of *completeProcessNormal* sets the *state* attribute to *completing* (cfr., $\mathcal{D}_{pistate}$ in Appendix A.) Furthermore, the right-hand side contains a subrule that applies the *iterated* construct to remove from the model all tokens that belonged to the completed process instance.

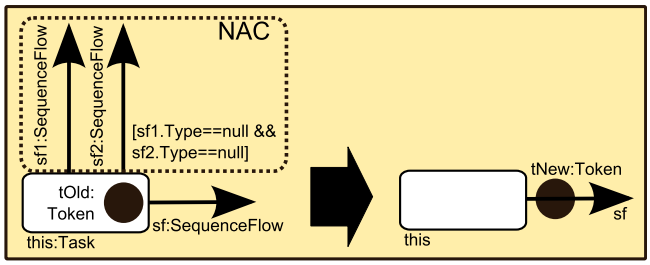


Fig. 14. Regular sequence flow for task elements (2/2): rule *leaveTaskOneOut*.

3.2. Activities

3.2.1. Sequence flow considerations

Fig. 13 shows the rule called *enterTask*. This rule handles the activation of task nodes regardless of whether such nodes have one or multiple incoming sequence flows. If a task node has multiple incoming flows, flows *without* a token do not prohibit the firing of this rule: the left-hand side only requires the presence of one link *sf* which has a token attached to it. Extra incoming sequence flows *with* a token can each produce firing of rule *enterTask* and can thus result in multiple tokens on the task node. The critical reader will notice that *enterTask* (Fig. 13) is quite similar to rule *enterExclusive* (Fig. 8). At the implementation-level, code duplication has been eliminated by means of reuse mechanisms but for the sake of simplicity we do not show these details in the visual rule representations.

Since it is quite trivial that throughout a process execution tokens remain within the same instance, we omit *Process Instance* nodes from both the left- and right-hand side in rule definitions unless such nodes are updated by the rewrite rule. For example, for Figs. 13–15, we omit the node and edges that ensure that *tOld* and *tNew* belong to the same process instance. In contrast, Fig. 16 does show the process instance node since this rule updates that node's *state* attribute and since non-trivial edges are connected to the node.

As a symmetric counter-part of rule *enterTask*, consider rule *leaveTaskOneOut*, which is visualized in Fig. 13. This rule formalizes the execution semantics for tasks that have one outgoing sequence flow, *sf*. Remark that *sf* can be a regular sequence flow, a conditional sequence flow, or a sequence flow with *otherwise* semantics (cfr., $\mathcal{D}_{flowtype}$ in Appendix A). The negative application condition (NAC) formalizes that the rule only applies in case there are not two (or more) regular outgoing sequence flows. Notice that the relation between *sf* and the elements in the NAC (*sf1* and *sf2*) is *homomorphic*[3]: the NAC expresses a pattern of two outgoing sequence flows (one of which is allowed to map to the same element as *sf*). The right-hand side of rule *leaveTaskOneOut* is straightforward: the token on the task node is destroyed and the outgoing sequence flow gets a new token.

Fig. 15 shows rule *leaveTaskMoreOut*. This rule complements rule *leaveTaskOneOut* by handling the case in which there *are* two (or more) outgoing sequence flows. This case realizes the so-called *AND split* workflow pattern [57], meaning that tokens must be put on all outgoing sequence flows. Its left-hand side contains a pattern with a *Task* node that has at least two outgoing sequence flows that are of type *null*. In the right-hand side, the token is removed from the task. Additionally, each regular outgoing sequence flow (i.e., those of type *null*) gets a new token assigned to it: since the *sf3* element is declared in an iterated subrule it can match multiple outgoing flows for one application of the *leaveTaskMoreOut* rule.

Note that variables *sf1* and *sf2* are allowed to be bound to the same element as variable *sf3*. In graph transformation rule jargon, for these variables we want homomorphic matching rather than (the default) isomorphic matching. Put differently, the implementation of this rule is annotated such that these pattern variables are allowed to be mapped *non-injectively* to host graph nodes.

3.2.2. Sub-process/call activity

Fig. 16 shows the rule called *enterSubprocess*, which formalizes subprocess invocations. The left-hand side consists of a *BlockActivity* node called *this*, which has an input sequence flow *sf* that is enabled (i.e., *sf* holds a token). The pattern also contains node *pi*, representing the process instance in which the token is contained. This node is shown explicitly since subprocess invocation involves side-effects at the process instance level. More specifically, in the right-hand side of the rule from Fig. 16, a new process instance node is created and relations to original process instance and the subprocess activity (i.e., *BlockActivity*) are established. Finally, *enterSubprocess* calls rule *reEnterSubprocess* and passes the subprocess activity (*this*) and instance (*newPI*) as arguments.

Rule *reEnterSubprocess* is shown in Fig. 17. Since variables *this* and *pi* are passed as rule parameters, they are already bound. By omitting their type information (i.e., *:FlowElementsContainer* and *:ProcessInstance*) in the left-hand side, these nodes will not be matched again to elements in the host. Rule *reEnterSubprocess* composes two subrules using the && operator. The subrule at the

[3] See Section 4.3.1 in the GrGen.NET manual [5]. Some graph transformation languages (such as Story Diagrams [16]) have a default isomorphic relation between negative pattern variables and their context variables. Users of such languages need to adapt some rules. For example, *leaveTaskOneOut* then requires just one outgoing sequence flow in its NAC block.

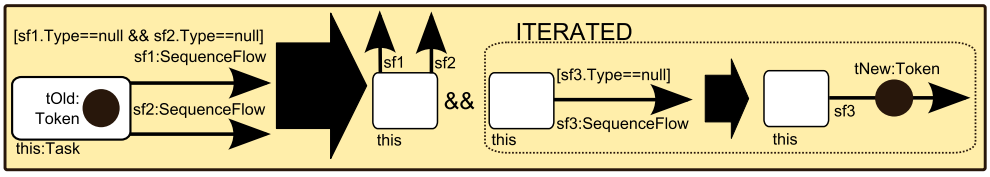


Fig. 15. *AND split* sequence flow for task elements: rule *leaveTaskMoreOut* .

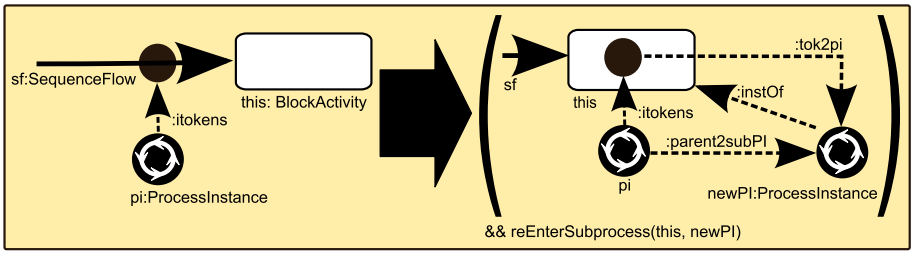


Fig. 16. Instantiation of subprocesses: rule *enterSubprocess*.

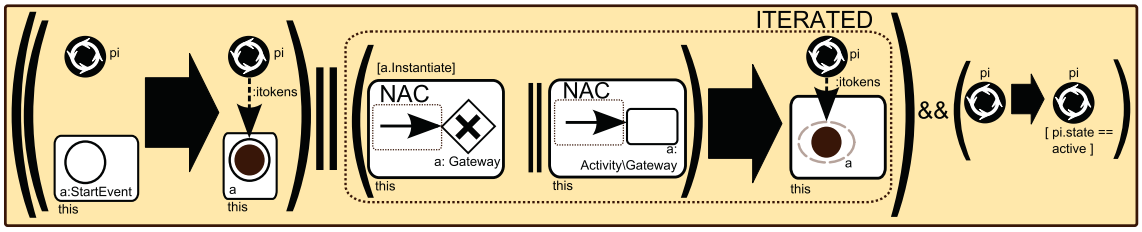


Fig. 17. Helper rule *reEnterSubprocess* with parameters *this* and *pi*.

right of the && operator sets the state of the process instance to *active*. The subrule at the left of the && operator (which is put between brackets) is further decomposed using the || operator.

The subrule on the left side of the || operator matches if the subprocess activity contains a start event *a* which does not yet contain a token for process instance *pi* (otherwise *pi* would be executing already). In case this subrule matches, a token will be added to *a* in the context of *pi*.

Due to short-cut evaluation of the || operator, the subrule on the right side will only be evaluated if the left-hand side fails. Therefore, it applies only in the situation where a subprocess activity *this* does *not* have a start event. For such subprocesses, the standard prescribes that certain elements without incoming arcs should receive a token: first of all, exclusive OR gateways (but only if their *Instantiate* attribute is set to *true*); secondly, all other activities (so for activities of another type than *Gateway* the *Instantiate* attribute is irrelevant). The ||-composed subrules in the *iterated* block ensure that all such elements receive a token upon subprocess instantiation.

Fig. 18 shows the rule called *leaveSubprocessNormal*. The rule matches when a process instance is in the *completing* state. Recall that rule *completeProcessNormal* (discussed in the context of Fig. 12) rewrites process instances to that state. Rule *leaveSubprocessNormal* complements the behavior of that rule by updating the process instance state to *completed* and by transferring control (i.e., a token) to outgoing sequence flows (see Figs. 19–21).

3.2.3. Loop activity

Fig. 22 shows the rule for entering a loop body. The rule applies to all activity types but subprocess activities. Fig. 23 shows the rule called *reEnterLoopSubprocess*. The rule realizes loop behavior for subprocess activities. Note that the configuration of the edges with

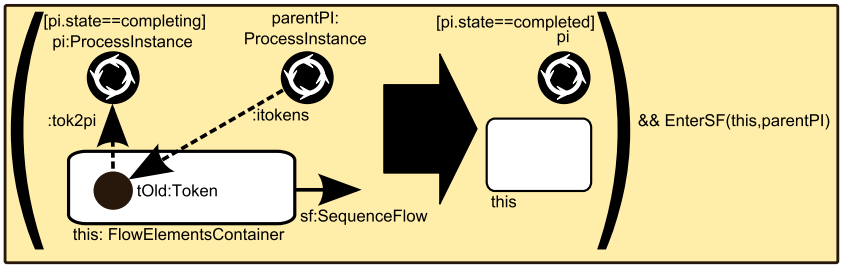


Fig. 18. Leaving a subprocess: rule *leaveSubprocessNormal*.

EnterSFone(this) || EnterSFandsplit(this)

Fig. 19. Helper rule *EnterSF (this:Activity)*. This helper rule is used in the definition of other rules than *leaveSubprocessNormal* too.

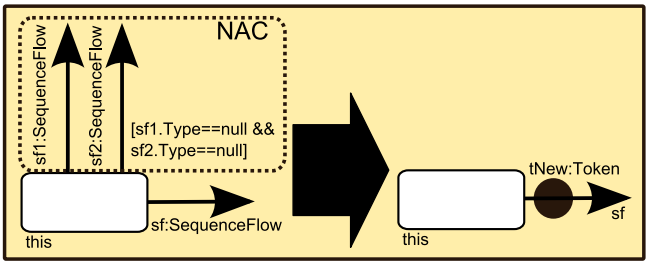


Fig. 20. Helper rule *EnterSFone (this:Activity)*.

respective types *InstOf, tok2pi* and *parent2subPI* is rather complex but this complexity follows directly from the BPMN standard. Fig. 24 shows the rule for skipping a loop body.

3.3. Gateways

We define the behavior of parallel, exclusive and inclusive gateways. Two rules define the behavior of the parallel gateway as shown in Figs. 25 and 26. Fig. 25 shows that a parallel gateway can receive a token in case it has no incoming control flows that do not have a token.

Two rules define the behavior of the exclusive gateway as shown in Figs. 8 and 27. Fig. 8 has already been discussed in Section 2.3. Fig. 27 shows what can happen when an exclusive gateway has a token: rule *leaveExclusive* can match in three cases. In any case, the token can be removed from the gateway. Additionally, a token can be put on one of the conditional outgoing control flows (case 1). If the gateway has a default flow, a token can be put on this default flow (case 2). If the gateway does not have a default flow, an exception can be generated (case 3). The latter case represents the situation in which there is no default flow and none of the conditions on the conditional outgoing flows are met.

Two rules define the behavior of the inclusive gateway as shown in Fig. 28 and 31. Fig. 28 shows when an inclusive gateway can receive a token. From a right-hand perspective, rule *enterInclusive* is quite similar to *enterParallel* and rather straight-forward in meaning. The left-hand side however requires more complex handling: according to the specification, an inclusive gateway can receive a token if it has at least one incoming sequence flow with a token (i.e., *sfWT* in Fig. 28) and if each sequence flow that does not have a token is not waiting for a token to arrive (i.e., such a sequence flow does not have a token upstream).

The requirement that an empty sequence flow should not have a particular token upstream is defined more precisely in the specification as: "There is no directed path from an upstream token to this sequence flow, unless:

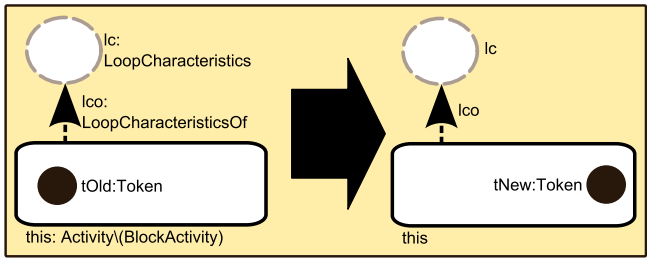


Fig. 22. Iterating a loop for a regular activity: rule *reEnterLoopActivity*.

- 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."

This part of the rule is realized by calling a subpattern called *HasTokenUpstream*. This helper pattern matches when there is a violating token upstream (i.e., a token that cannot flow down to non-empty incoming sequence flows of the gateway). By applying this helper pattern in a NAC, rule *enterInclusive* ensures that there are no such violating tokens. Helper *HasTokenUpstream* walks the sequence flow graph upstream by taking the source activity of its parameter *sf*.

By constraining that *a* is different from *gw*, the pattern realizes the first exception (i.e., *"unless: ... the path visits the inclusive gateway"*). Then, the pattern checks three cases of possible violation:

- either the activity *a* holds a token, or
- an incoming sequence flow of *a* holds a token, or
- a transitive successor upstream matches this pattern.

Recall that the token should only be classified as violating if from activity *a* there is no directed path to a non-empty incoming sequence flow. This additional condition is checked by the NAC at the bottom of Fig. 29. The NAC applies *DirectedPathBetween* whose definition is shown in Fig. 30.

Fig. 31 shows what can happen when an inclusive gateway has a token: rule *leaveInclusive* is quite similar to rule *leaveExclusive* (cfr., Fig. 27). Similarly to rule *leaveExclusive*, it consists of three cases: one for activating the conditional control flows, one for activating the default control flow and one for generating an exception in case there is no default control flow and none of the conditions on the conditional outgoing flows are met. However, instead of only putting a token on one conditional outgoing flow, an inclusive gateway can put a token on any number of its conditional outgoing flows. This is represented in the rule by the *iterated* block annotated by 1..*n*. This annotation represents that the iterated block should be applied to a number of occurrences between 1 and *n*. The value of *n* is set to the number of conditional outgoing flows of the gateway (i.e. the number of times that the subpattern *P* can be matched).

As indicated by Fig. 10, the behavior of leaving an inclusive OR gateway is also formalized for *Task* activities. More specifically, if a task has more than one outgoing conditional sequence flow, it

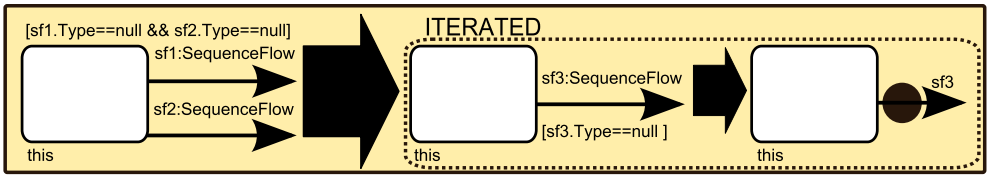


Fig. 21. Helper rule *EnterSFandSplit (this: Activity)*.

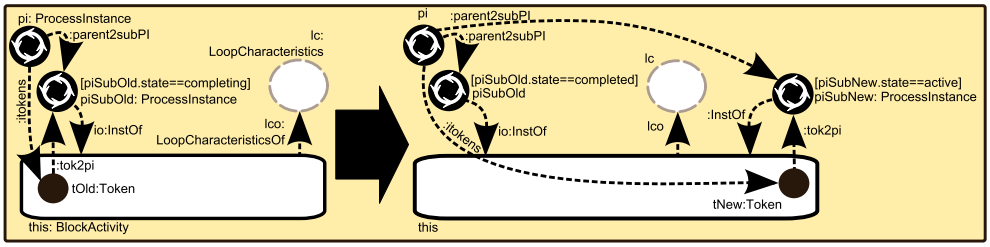


Fig. 23. Iterating a loop for subprocess activities: rule *reEnterLoopSubprocess*.

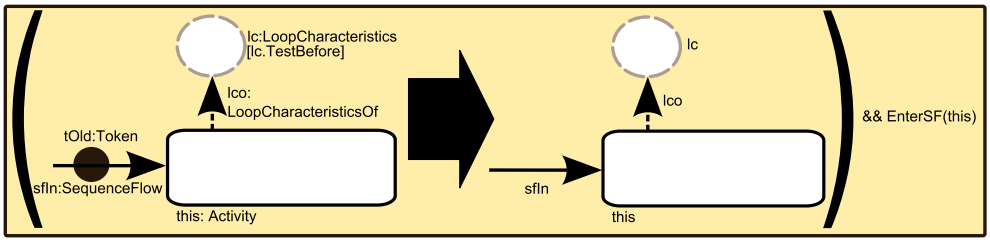


Fig. 24. Skipping a *while-do* loop for regular *or subprocess* activities: rule *skipLoopActivity*.

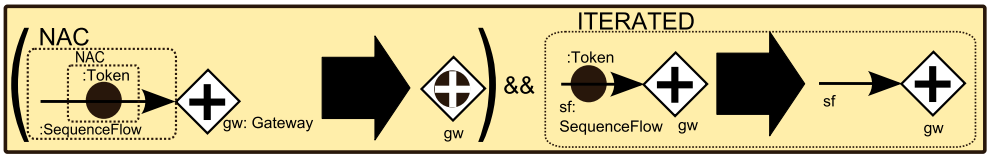


Fig. 25. Enter a parallel gateway: rule *enterParallel*.

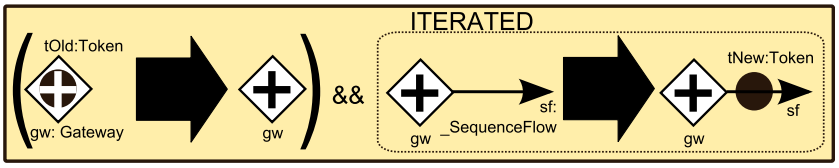


Fig. 26. Leave a parallel gateway: rule *leaveParallel*.

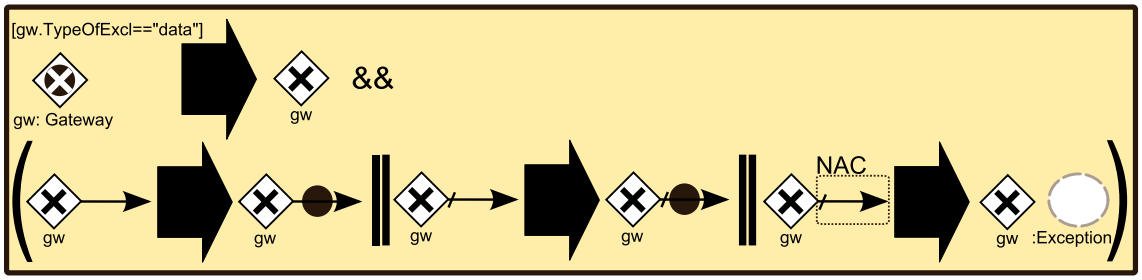


Fig. 27. Transfer control from an XOR-split: rule *leaveExclusive*.

matches rule *leaveImplicitInclusive*. The latter rule is not discussed here but is straightforward as it can reuse the right-hand side of rule *leaveInclusive*.

Rule *catchImplicitlyThrownException* (shown in Fig. 32) formalizes a proposed extension to the BPMN 2.0 standard. Without this rule, exceptions that are thrown by (X) OR splits, will never be handled. We propose to support the handling of such exceptions by means of a boundary intermediate error event without an error-code (giving it the expected catch-all semantics). Remark that the NAC avoids a conflict with rule *leaveThrowErrorEvent* (cfr., Fig. 38).

3.4. Events

There exist four basic types of events: start events, intermediate events, intermediate boundary events and end events. The basic

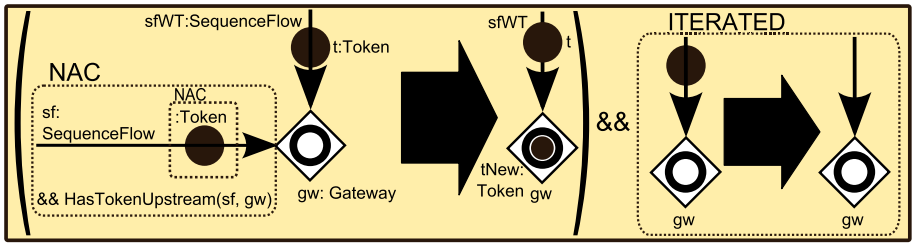


Fig. 28. Enter an inclusive gateway: rule *enterInclusive*.

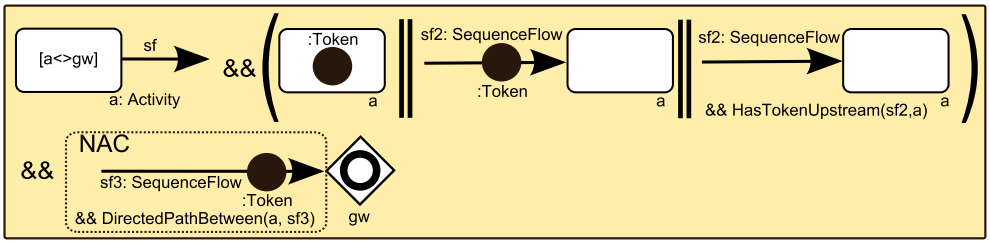


Fig. 29. Helper pattern *HasTokenUpStream (sf: SequenceFlow, gw: Gateway)*.

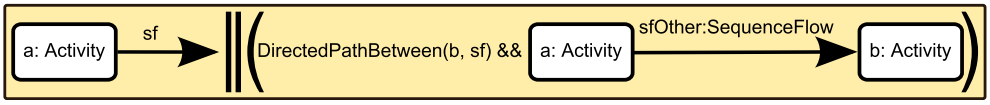


Fig. 30. Helper pattern *DirectedPathBetween (a: Activity, sf: SequenceFlow)*.

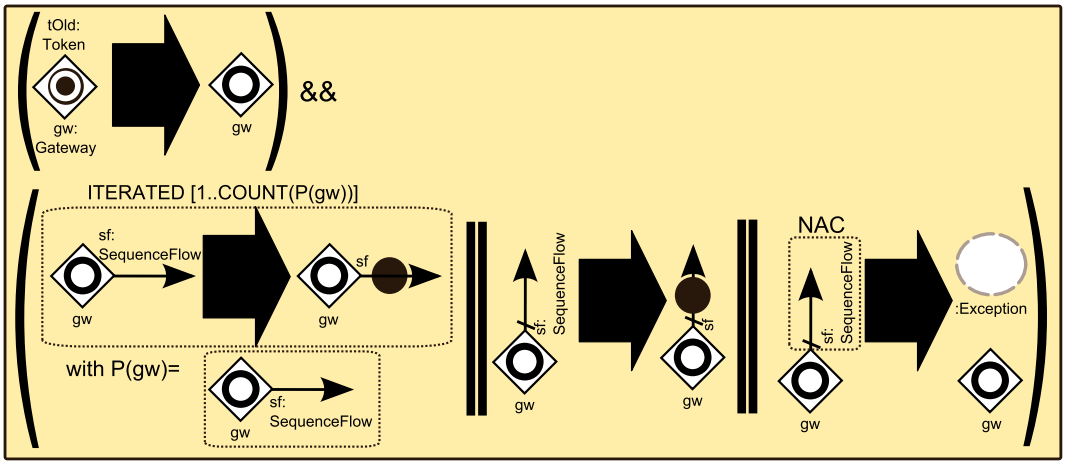


Fig. 31. Leave an inclusive gateway: rule *leaveInclusive*.

behavior of start events is explained in the Section 3.1 on instantiation. The basic behavior of an intermediate event is the same as for a task. The basic behavior of an intermediate boundary event is that it can fire while the activity on whose boundary it is, is active. There exist two variants of this behavior. One in which the boundary activity interrupts the activity and one in which the activity can continue. We have formalized the first variant in rule *enterAutonomousBoundaryEvent* (cfr., Fig. 33). The basic behavior of an end event is to simply receive a token. This is formalized by rule *enterEndEvent* (cfr., Fig. 34). Note that this rule applies not only to regular end events (i.e., those of result type *none*), but also to message events, etc.

Events can catch triggers (cfr., rule *enterMessageCatch* in Fig. 35) that can be thrown by events (e.g., rule *leaveMessageThrowEvent* in Fig. 36). The BPMN 2 standard defines a number of trigger types which we have formalized in Appendix A (cfr., $\mathcal{D}_{trigtype}$). We define the execution semantics for the message, error, compensation and signal events below. Some trigger types require support of technical infrastructure at process deployment time. An example of this is the *timer* event that triggers when a preset moment in time is reached. Since our formalization aims at the business level, it does not rely on system-level clocks or other infrastructure facilities. Instead, timer events can fire non-deterministically. The same holds for *conditional*, *cancel*, *multiple*

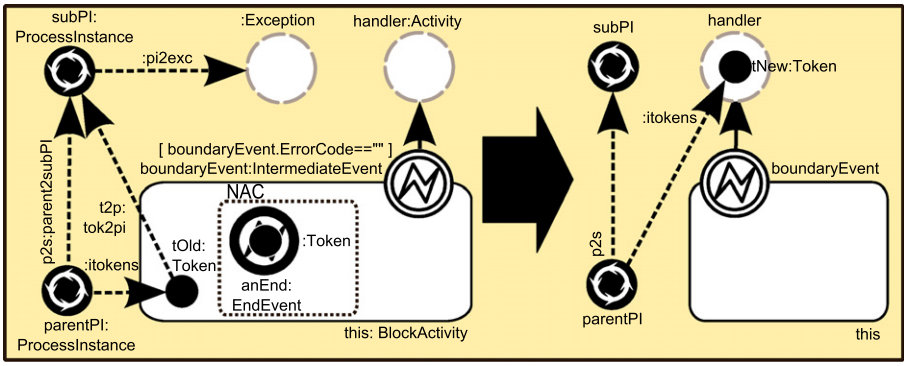


Fig. 32. Handling exceptions that are thrown implicitly (i.e., *not* by a throw event).

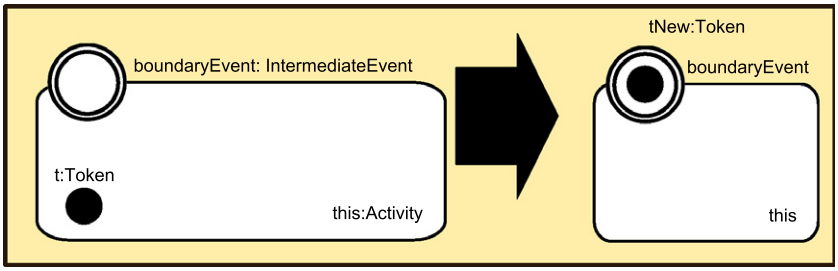


Fig. 33. Enter an autonomous boundary event: rule *enterAutonomousBoundaryEvent*. Note that unlike most rules, the removal of token *t* represents a true element *deletion* here, since the activity is interrupted before completion and therefore token *t* should not be preserved in a "undo history".

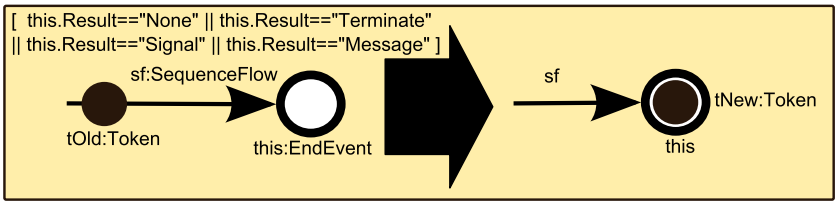


Fig. 34. Entering various types of end events: rule *enterEndEvent*.

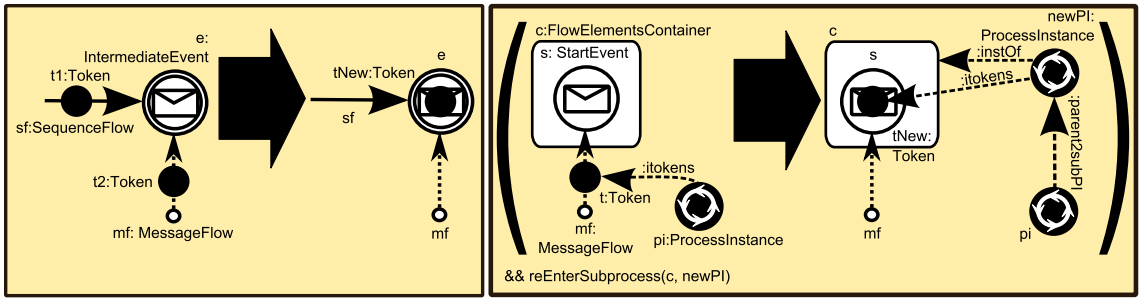


Fig. 35. Rule *enterMessageCatchIntermediateEvent* and *enterMessageStartEvent*.

and *none* events (which by definition do not have a specific semantics).

When an end event throws an event of type *error*, the related process instance will enter a failed state. Fig. 37 shows rule *enterThrowErrorEvent* which formalizes this behavior. Since the BPMN 2 standard does not support throwing errors from *intermediate* events, we provide no rule for raising errors from those events. Fig. 38 shows rule *leaveThrowErrorEvent* which formalizes the BPMN 2 error handling support. In case the subprocess in which the error occurs has a catch error event attached to it (cfr., the

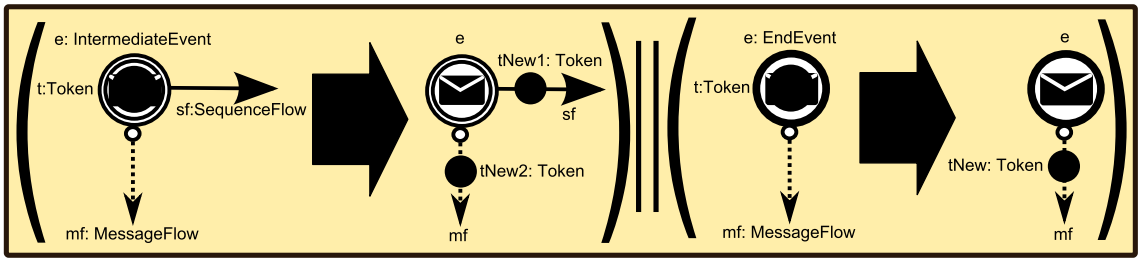


Fig. 36. Throwing a message: rule *leaveMessageThrowEvent*.

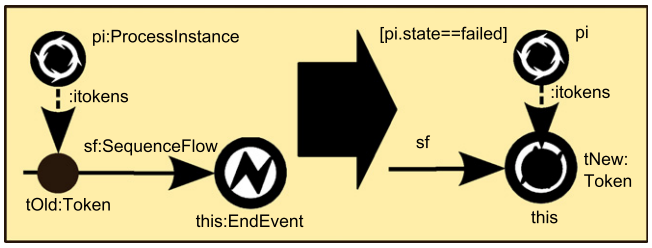


Fig. 37. Throwing an error: rule *enterThrowErrorEvent*.

optional block in Fig. 38), a token is put on the activity that this event points to. The catch error event must either have the same error code as the throw error event, or it must have no error code, in which case it reacts to all error events. Also refer to the discussion of Fig. 32 for another case of *catch-all* behavior.

3.4.1. Compensation events

Figs. 39 and 40 show the rules for compensation behavior. The right-hand sides of these rules is the same: besides the trivial move of the token from the input sequence flow to the event node, both rules delegate to helper *AddUndoTokens* whose definition is shown in Fig. 41. That rule has two parameters: the element *thrower* that throws the compensation event, and the process instance *pi* that needs to be compensated.

AddUndoTokens adds special tokens (elements of type *UndoToken*) to the activities that need to be compensated. There are two cases for adding such tokens. Both cases are handled by alternative subrules. Note that the upper part of Fig. 41 clearly shows that the subrules are complementary: the two left-hand sides shown there contain the same pattern (a pattern with edge type *activityRef*) but it is negated (using a NAC) in the left-hand side of the right-most subrule.

The first subrule (before the || operator) deals with the compensation of one specific activity. Following the BPMN standard, this corresponds to the situation where there is a link of type *activityRef* between the throwing event and the to be compensated activity. In this case, only the activity that needs to be compensated gets an *UndoToken*.

The second subrule deals with the opposite case (i.e., the case where no activity has been modeled for explicit compensation handling). In that case, the BPMN 2 standard prescribes the implicit compensation of all completed activities from the current subprocess as well as from recursively spawned child processes. The subrule also sets the status of the involved process instances to *compensating*. This is shown at the top right of Fig. 41. The recursive compensation behavior realized by the two nested *iterated* blocks that are shown at the bottom of Fig. 41. The outer *iterated*

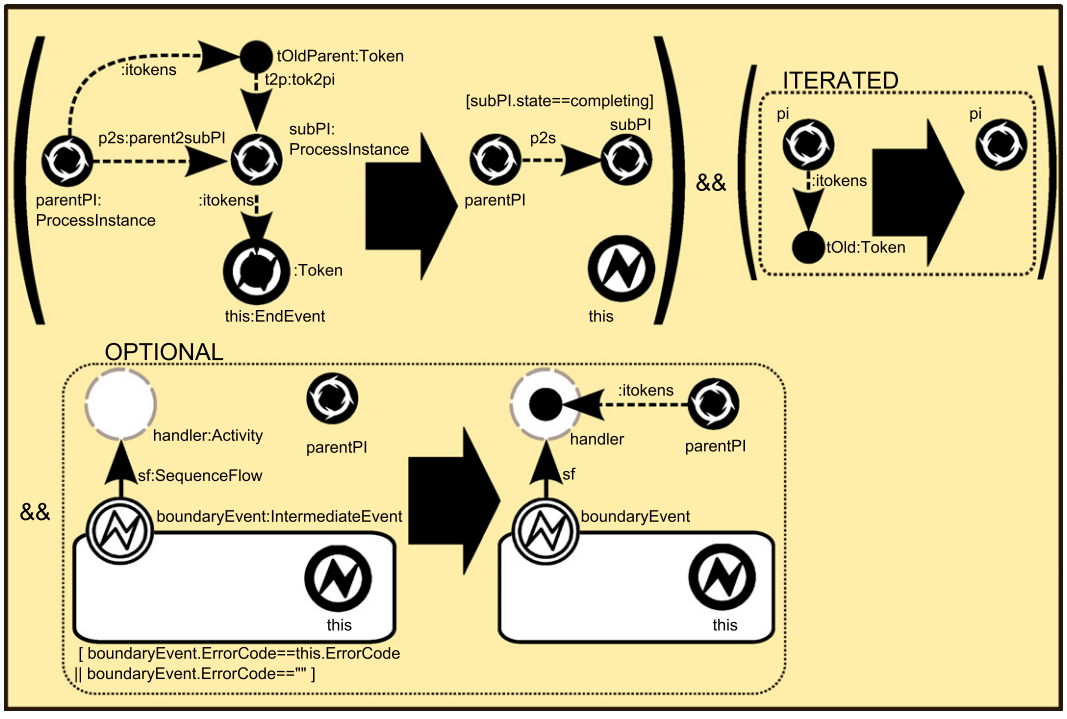


Fig. 38. Handling errors: rule *leaveThrowErrorEvent*.

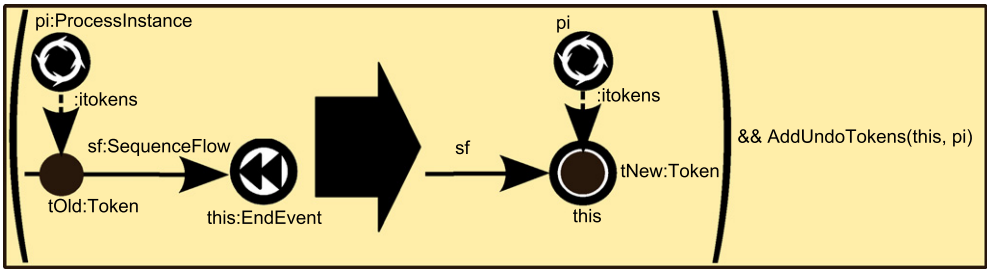


Fig. 39. Compensation (1/2): *enterCompensationEndEvent*.

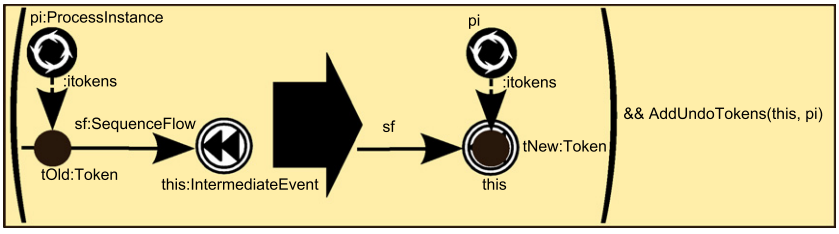


Fig. 40. Compensation (2/2): rule *enterCompensationThrowIntermediateEvent*.

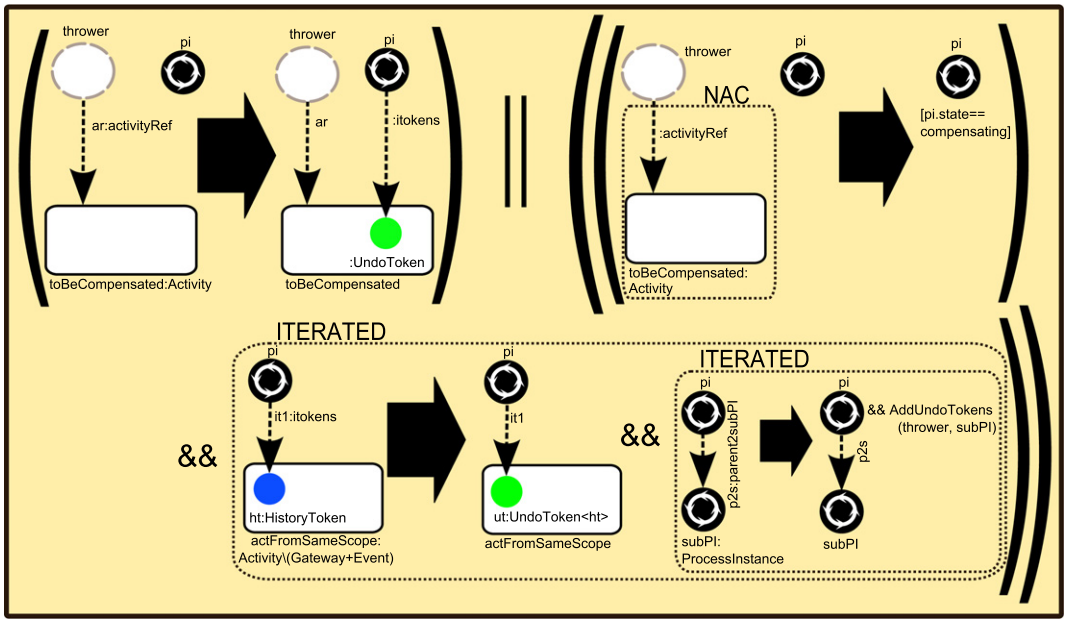


Fig. 41. Recursive helper that produces *Undo* tokens: rule *AddUndoTokens*.

block matches all so-called *history* tokens in the context of process instance *pi*. As indicated by the rewrite arrow in this block (and expression *ut*:*UndoToken*<*ht*> in particular) each matching history token should be retyped to an *undo token*. This realizes the compensation of all completed activities at one process instance level. Within this *iterated* block, there is a second *iterated* block. The latter block matches each subprocess *subPI* of the instance *pi*. For each such subprocess, rule *AddUndoTokens* is executed recursively.

The use of *HistoryToken* nodes requires some further explanation, especially since a proposed addition to the BPMN standard model elements. History tokens are created in all cases where a regular *Token* node is deleted by our rules. More precisely, every delete operation on a node of type *Token* is replaced by a node re-type operation, from type *Token* to type *HistoryToken*. For all rules so far, the effect of the re-typing is the same as the effect of a real delete operation, since the *Token* nodes (1) are no longer visible in BPMN concrete syntax, and (2) will no longer match in the left-hand sides of our the rules that have been discussed so far. By keeping a history of tokens that were conceptually removed by these rules, the underlying graph has a notion of which activities have been completed. Without such history information, it would be impossible to realize compensation behavior.

Figs. 42 and 43 show the rules that formalize the actual compensation behavior. Rule *UndoProcessInstance* matches a top-most process that contains an *UndoToken*. As explained in the context of Figs. 39 and 41, such tokens represent ongoing compensation for activities that had completed. Rule *undoProcessInstance* matches those process instance *pi* that need to be undone. The rule

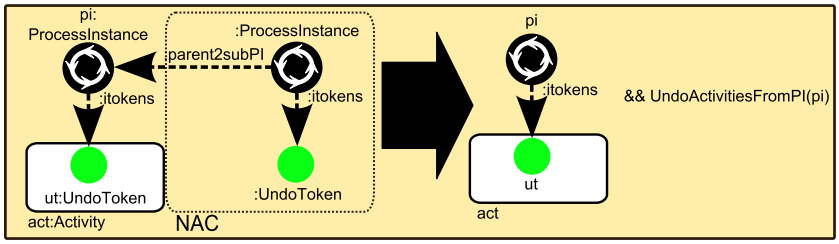


Fig. 42. Triggering the actual compensation behavior: rule *UndoProcessInstance*.

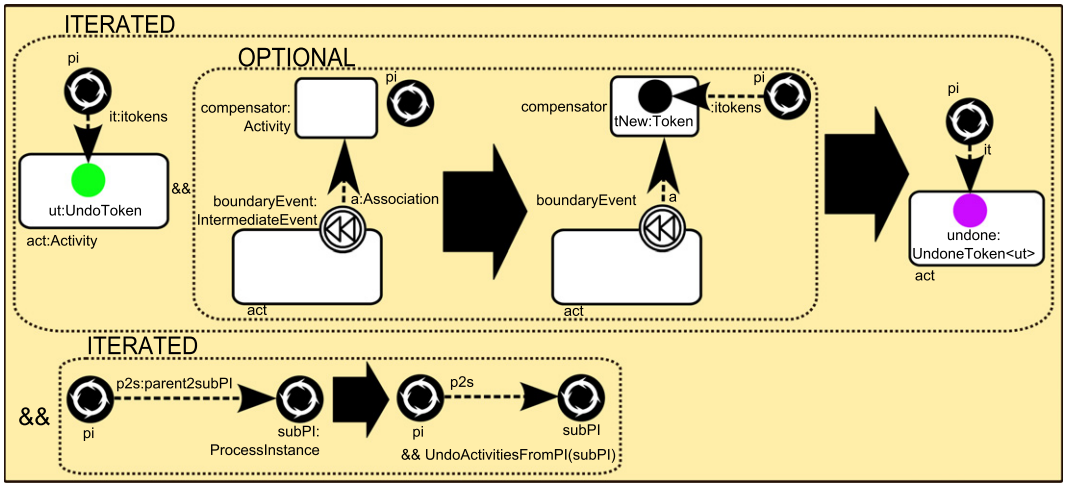


Fig. 43. Helper of *UndoProcessInstance*: rule *UndoActivitiesFromPI*.

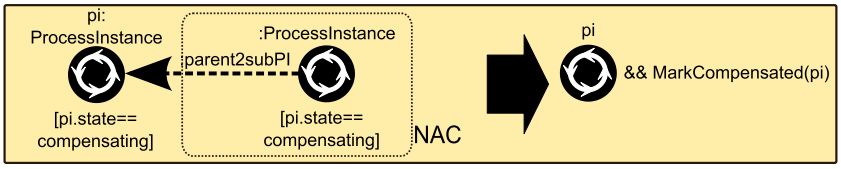


Fig. 44. Compensation completion: rule *completeSubprocessCompensation*.

delegegates to helper rule *UndoActivitiesFromPI* in its right-hand side. That helper rule recursively updates the tokens in *pi* and its child subprocesses. Since the recursion goes top-down (i.e., from parent to child subprocess), the NAC in the left-hand side of *UndoActivitiesFromPI* ensures that no parent process of *pi* could produce a match as well.

Helper rule *UndoActivitiesFromPI* models the compensation of each individual activity *act* in process instance *pi*. As indicated by transformation variables *ut* and *undone*, this involves the re-typing of a node of type *UndoToken* to a node of type *UndoneToken*. Moreover, as indicated by the embedded *optional* block, this may involve the activation of a compensation activity. Such compensation activities can be present as the targets of *Association* edges that originate from a boundary intermediate compensation event of *act*.

Rules *completeSubprocessCompensation* and *MarkCompensated* (cfr., Figs. 44 and Fig. 45) formalize the completion of compensation behavior. Similar to rules *UndoProcessInstance* and *UndoActivitiesFromPI*, they apply a NAC and recursion to handle a complete hierarchy of process instances atomically. In accordance with the BPMN standard, *MarkCompensated* updates the *state* attribute of its process instance to *compensated*. Additionally, the rule removes tokens (if any) from compensation activities. This removal of tokens is in fact not prescribed by the BPMN specification and therefore it was initially not part of our formalization. While testing our rules on a comprehensive testsuite of process models, we have discovered however that tokens that are added to these compensation activities (e.g., by the *optional* subrule of rule *UndoActivitiesFromPI*) always prevent in a counter-intuitive way the proper termination of processes (see [20] for a formal definition of the term *proper*).

To make the effect of the various rules for compensation more concrete, we illustrate them on the simple example process shown in Fig. 46. The example is taken from a professional weblog[4] and relates to the recision of a sales transaction. The discussion on the weblog is a typical case of semantical confusion from practitioners, which can be resolved by means of formalization and a reference implementation. Section 5 even shows how our implementation facilitates the understanding of a more complex extension of this example model.

Fig. 47 shows the statespace for the (relatively) simple model. Markings 1 to 8 are rather straigth-forward: they encode a simple

[4] http://tynerblain.com/blog/2006/09/18/bpmn- compensation-correction/.

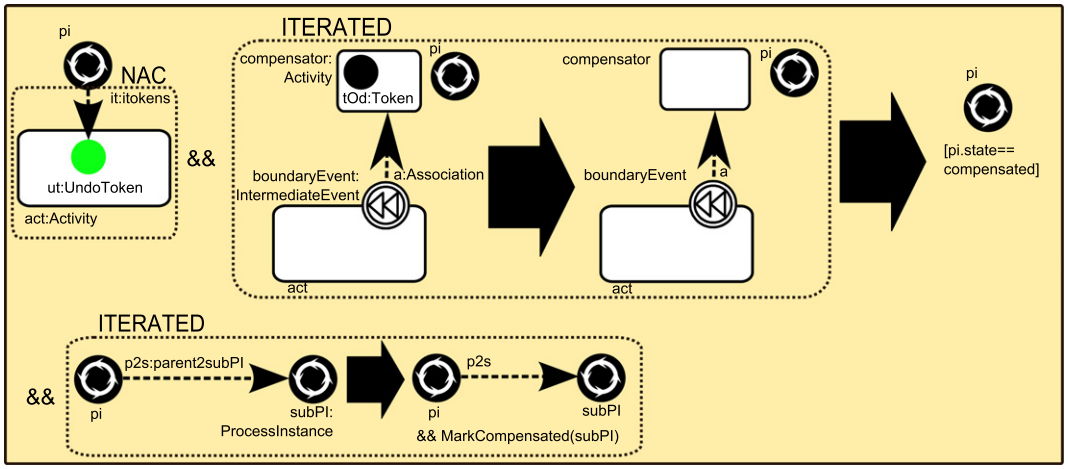


Fig. 45. Compensation completion: helper rule *MarkCompensated*.

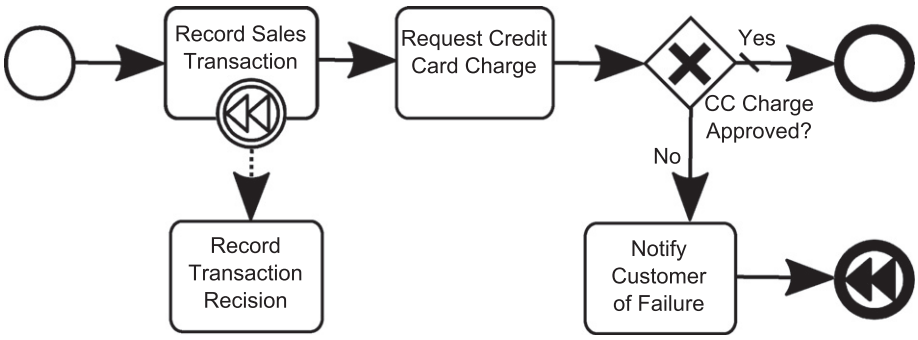


Fig. 46. Sales recision process which has triggered discussion between practitioners.

sequential activation of the start process, task *Record Sales Transaction*, task *Request Credit Card Charge* and finally the XOR split.

After marking 8, the process can go to marking 9 or 13. Marking 9 represents the state that is reached when following the sequence flow labeled as *Yes* while marking 13 represents the state when following flow *No*. After moving a token over activity *Notify Customer of Failure* (markings 14 and 15), the process reaches the compensation end event. In this example process, the compensation trigger does not have a specific activity reference, which means that the complete process needs to be compensated. Fig. 48 shows the details of markings 15–18 in order to clarify the effect of rules related to compensation handling. Additionally, the transition from marking 15 to 16 clarifies how normal tokens are transformed into history tokens: marking 15's token on the flow to the compensation end event has become part of the history in marking 16. This is the effect of applying the side-effects of rule *enterCompensationEndEvent* (cfr., Fig. 39). Recall that the delete of *tOld* is in fact a re-type from *Token* to *HistoryToken*.

The effect of the delegation to the recusive helper *AddUndoTokens* (cfr. Fig. 41) is quite straightforward for the transition from marking 15 to 16: since the compensation end event in the running example does not have an outgoing *activityRef* link, the leftmost subrule of *AddUndoTokens* (i.e., before operator ||) fails. Therefore, the complementary rightmost subrule (including the NAC) does match. As indicated by the top-right expression in Fig. 41, this leads to an update of the *state* attribute of the corresponding process instance. Also, the two nested *iterated* blocks after the && operator are evaluated. The outermost *iterated* block matches with variable *ht* bound to the three tokens that are in *Task* activities in marking 15 (cfr., Fig. 48). The right-hand side of this subrule changes the type of these three tokens to *UndoTokens*, as illustrated by marking 16 in Fig. 48. The nested *iterated* block does not match for on marking 15 since the process instance has no subprocesses attached to it.

As indicated by the statespace shown in Fig. 47, the transition from marking 16 to 17 is realized by an application of rule *UndoProcessInstance* (cfr., Fig. 42). The left-hand side of that rule matches with variable *pi* bound to the unique process instance from our running example, since (1) indeed there is at least one *UndoToken* in that process instance and (2) the process instance has no parent process instance which satisfies the same property (so the NAC is satisfied). The right-hand side of *UndoProcessInstance* delegates to helper rule *UndoActivitiesFromPI* and passes the unique process instance of the example as an argument. Consequently, the two *iterated* blocks shown at Fig. 43 are executed with *pi* bound to that unique process instance. In the first *iterated* block, *ut* is bound to the three *UndoToken* instances shown in marking 16 of Fig. 48. Also, *act* is bound three times as well (to the activities holding the *UndoToken* instances). One of these three activities has a compensation handler attached to it and therefore the *optional* block of *UndoActivitiesFromPI* matches for just one of the three activities. Due to the right-hand side of the *optional* subrule, that one activity's compensation handler receives a token (bound to variable *tNew* in the right-hand side of the subrule). Due to the right-hand side of the first *iterated* block, all *UndoToken* instances are retyped to *UndoneToken*. Finally, the second *iterated* block of *UndoActivitiesFromPI* is executed. Due to the lack of subprocesses, the subrule in that block does not match.

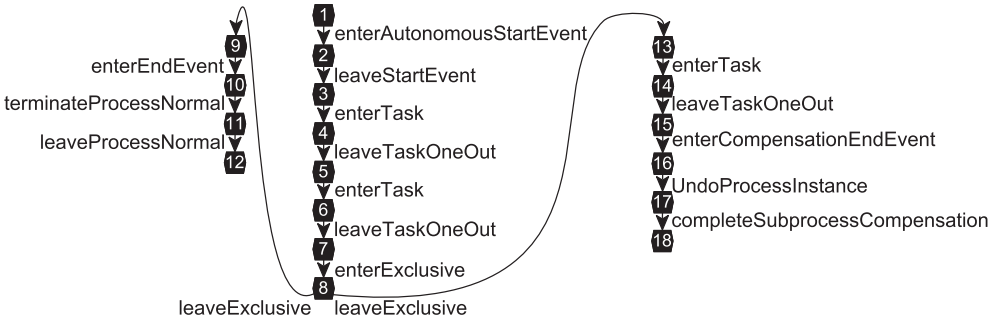


Fig. 47. Statespace for the sales transaction recision process.

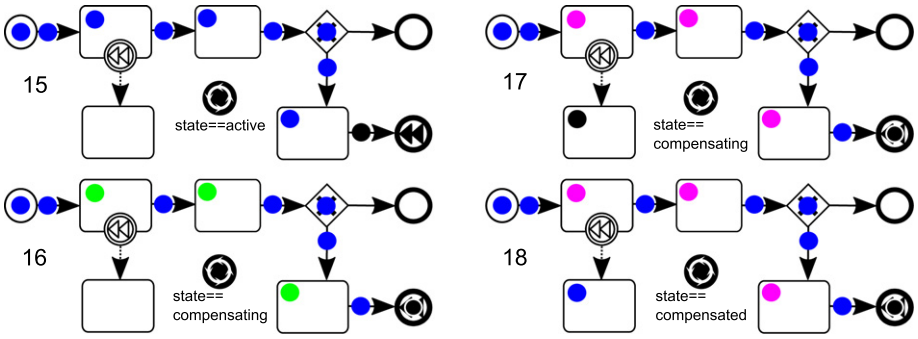


Fig. 48. Marking details for the compensation aspects of the sales recision process.

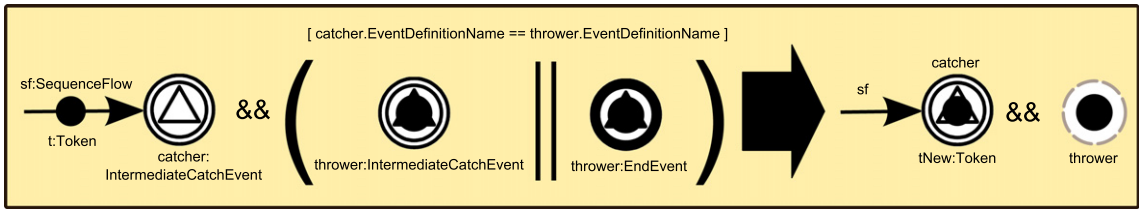


Fig. 49. Reacting to a signal: rule *enterSignalCatchIntermediateEvent*.

3.4.2. Signal events

Signal events can be thrown by signal intermediate events or by signal end events. Subsequently, they can be caught by a signal catch intermediate event. It is important that the signal catch event is "listening" (i.e., there should be a token on its incoming sequence flow). When no signal catch events are listening, the signal event can be lost. Figs. 49 and 50 show these two possible situations.

Rule *enterSignalCatchIntermediateEvent* (cfr., Fig. 49) shows that an intermediate signal catch event can be entered in case there is a corresponding signal throw event with a token. The throw and the catch event are said to be *corresponding* if both refer to the same signal definition. Remark that in the right-hand side, the signal catch event receives a token, but the token on the corresponding thrower is *not* removed. If the token would be removed, then at most one catch event could be activated by a signal. By leaving the token on the thrower, rule *enterSignalCatchIntermediateEvent* can fire many times (i.e., once for every corresponding catch event with an enabled input sequence flow). Also remark that such multiple firings will produce separate markings. This reflects that our formalization does not impose that all catch events react at the same time (i.e., responding to a common signal does not synchronize concurrent threads). Obviously, once (and only once) all corresponding catch events have been activated, the token on the throw event should be removed. This is realized by rule *leaveSignalThrowEvent*.

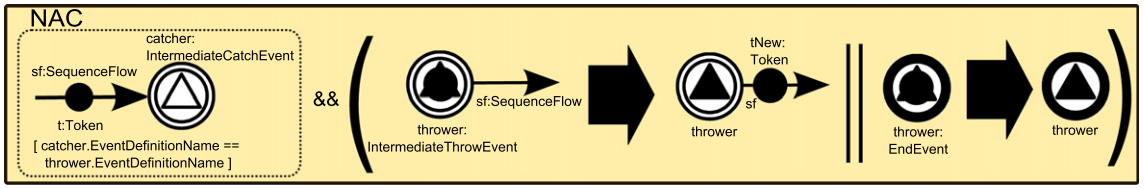


Fig. 50. Dropping a signal: rule *leaveSignalThrowEvent*.

Fig. 50 shows that a signal can be lost in case there is a signal throw event with a token, but there is no corresponding catch event. The rule simply removes the token in the case of an *end* event as a thrower, while in the case of an *intermediate* event as a thrower, the outgoing sequence flow receives a token. Remark that *leaveSignalThrowEvent* can fire in two scenarios: first of all, the rule can fire in case the receiver side has not yet reached the catch event for processing the signal. Secondly, the rule can fire after one or more receivers have received the signal event. In the former case, the signal has been lost. In the latter case, the signal has resulted in an activation of all corresponding catch events. In summary, the proposed transformation rules formalize signal broadcasts that are reliable, but that will get lost for those that are not listening to the broadcast channel.

4. Implementation

This section discusses the implementation of the graph transformation rules from Section 3. Section 4.1 presents the reference implementation of our rules, Section 4.2 highlights interesting features of the supportive GrGen.NET platform, Section 4.3 revisits the traceability aspect and reflects on which level of traceability is desirable at the implementation level. Finally, Section 4.4 discusses possible alternative support platforms.

4.1. Implementation in GrGen.NET

The reference implementation is accessible from a web-based front-end as well as through various local GrGen.NET scripts [59,9]. It supports the following user scenarios:

Manual execution. In this scenario, the user can simulate a BPMN 2.0 model, by explicitly choosing at any time (a) which rule to evaluate, and (b) in the case the selected rule has multiple matches: which match to apply. No statespace is built in this mode. Instead, the execution produces just one sequential trace.
Batch statespace generation. In this scenario, the transformation rules are executed non-deterministically for a given number of iterations, such that they generate a statespace. More specifically, it collects which markings are reachable from which other markings, by executing which transformation rule. The statespace can be used for various forms of statespace exploration, but should be used with the caution that the statespace that is generated is not necessarily complete.
Interactive statespace generation. This scenario supports the interactive extension of partial statespaces. This is useful in the case that the statespace for an input model is very large (or even infinite). Users can then manually explore particular paths further.

The web-based front-end supports the first user scenario and is intended for illustration purposes only. The GrGen.NET scripts support all three scenarios. They are made available through an online virtual machine that contains (1) the GrGen.NET implementation of the rules from Section 3, (2) execution scripts for each of the three user scenarios outlined above, (3) a large collection of test models, (4) the version of GrGen.NET that should be used with the implementation, (5) an XPDL based BPMN editor, and (6) a tool to import XPDL into GrGen.NET. Note that the scripts are based on the GrGen.NET debugger and therefore they do not visualize BPMN models in exact BPMN notation. However, due to various configuration mechanisms, the graphs in the debugger do resemble that notation to a large extent.

Fig. 51 shows an application of scenario one (*Manual Execution*), for the execution of a process that contains an embedded subprocess activity followed by an implicit AND split. The manual execution has reached the state right after the termination of the subprocess. This can be seen on the visual debugger window shown at the left of the figure, where both sequence flows that follow the embedded subprocess have a token. The debugger terminal on the right shows that the user can choose at this point which match the rule *enterTask*. By pressing "0", or "1", the user can cycle between the two matches (i.e., he can choose between continuing execution of the process in the left or the right branch). By pressing other keys, the user can let the engine continue non-deterministically, or try to evaluate a particular rule: pressing "n" continues

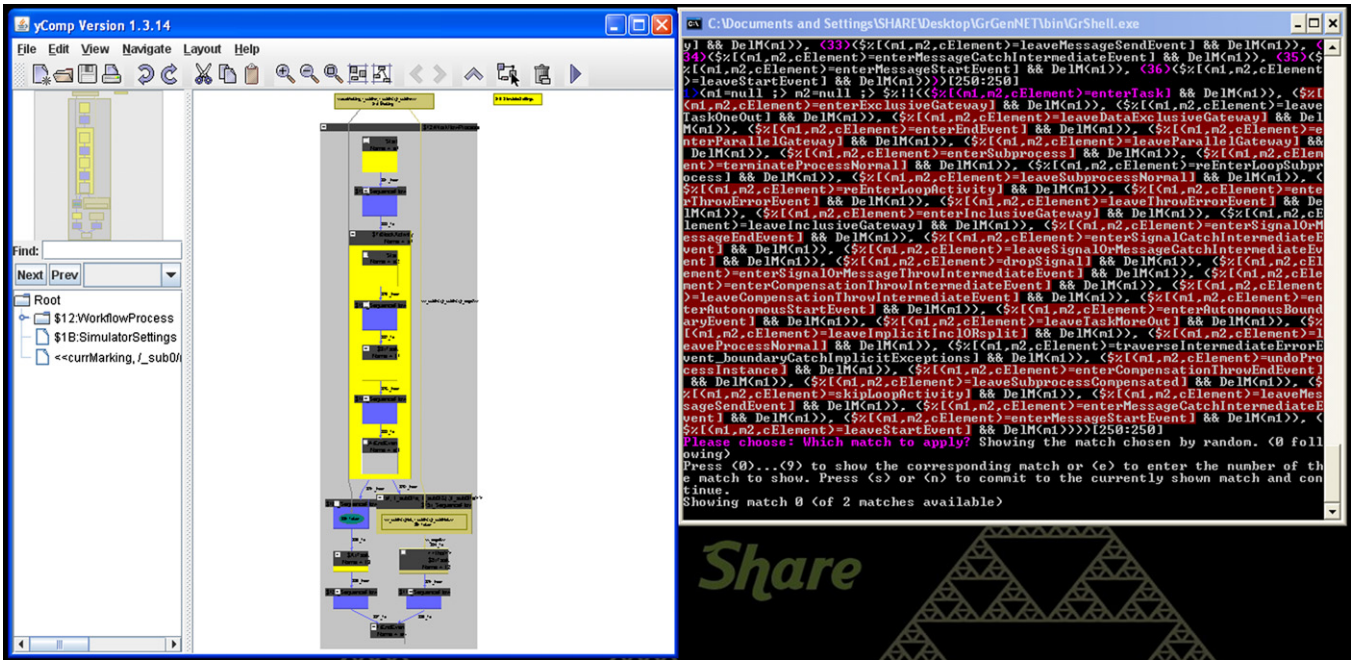


Fig. 51. Screenshot of the *Manual Execution* mode.

with a random match. Afterwards, pressing "e" will enable the user to select which rule to fire next. Pressing "o" will fire a rule non-deterministically (and automatically ignore all rules that do not match at this point).

The screenshot shown in Fig. 51 is the result of pressing "o" in the state where both the sequence flows that follow the *BlockActivity* contain a token. As indicated by the red markers in the right of the figure, various rules (such as *enterExclusiveGateway* and *leaveExclusive*) have failed to match. The cyan text "enterTask" indicates that rule *enterTask* is applicable twice in the current version of the host graph. In fact, the GrGen.NET debugger has interrupted the rewriting and is prompting for user input. This behavior is the result of the script statement $%[enterTask]: the GrGen.NET script operator [ ] instructs the engine to look for all matches of the rule "enterTask". By prefixing that [ ] operator by a $ sign, the script instructs the engine to rewrite just one match. Finally, the % symbol (leading to $%[ ]) instructs the engine to let the user choose which particular match to rewrite.

Fig. 52 shows an application of scenario two (*Batch Statespace Generation*). This example involves a very simple process, consisting of just six activities (a start and an end event, two parallel gateways, and two tasks that run concurrently between these gateways). The statespace for this example contains 18 markings. The GrGen.NET script for constructing the statespace executes about three seconds on a virtual machine with 1 GB of main memory and a mainstream CPU at the time of writing [59].

Fig. 52 shows the result of a script that visualizes just one marking as an overlay on the BPMN diagram. Again, the script applies an operator for retrieving user input: the statement *h1=$%(Marking)* lets the user select one node of type *Marking*. The script takes the selected node and then applies some helper rules for only showing the tokens from the selected marking. In this case, we have selected the *Marking* node that is pointed at as "1" in the figure. This marking contains two tokens, both of which are pointed at as "2" in the figure. In this specific case, the tokens reside on the two concurrent tasks. The figure shows that the statespace branches and then merges again in a common node. This is since the order of completion of tasks is arbitrary in concurrent paths but after two steps, the second parallel gateway should hold a token.

The third execution mode (*Interactive Statespace Generation*) looks like a combination of Figs. 52 and 51: after executing the operational semantics rules for a specific number of iterations, users run the rewriting system interactively and select a specific transformation rule using the mechanisms shown in Fig. 51.

4.2. Discussion of the GrGen.NET implementation

Based on our experience with the implementation of the execution semantics, we have identified the following strengths of the GrGen.NET platform:

Visual debugger. GrGen.NET includes a visual debugging tool. We have used this tool to hunt down bugs by stepping through the execution of complex test models. However, as illustrated in Section 4.1, we also still use the tool to execute BPMN models now that the implementation is stable. A major advantage is that the visualization can easily be changed to tune the level of detail. For example, in order to also visualize *ProcessInstance* nodes, just one configuration line needs to be adjusted. Remark that when no dedicated graph transformation tool is used at all, the debugging support of a general programming language (such as Eclipse for Java) is significantly more low level: not only would support for visual debugging on the BPMN model be lacking, the rule-oriented perspective is missing in such environments.

Fast. The GrGen.NET engine implements various domain-independent optimizations. Although the tool was originally developed for the domain of compiler construction, various benchmarks have illustrated excellent performance in other domains (including for example the execution of huge Petri nets [19,6,30]). Van Gorp and Eshuis have demonstrated that the advantage of GrGen.NET over a general purpose programming language is that better performance can be achieved without writing performance related code for a specific case study [60].

Actively maintained. Since GrGen.NET is still a research prototype, it is of uttermost importance that problems with the engine are solved in a timely manner. For the implementation work related to this paper, we have encountered some bugs

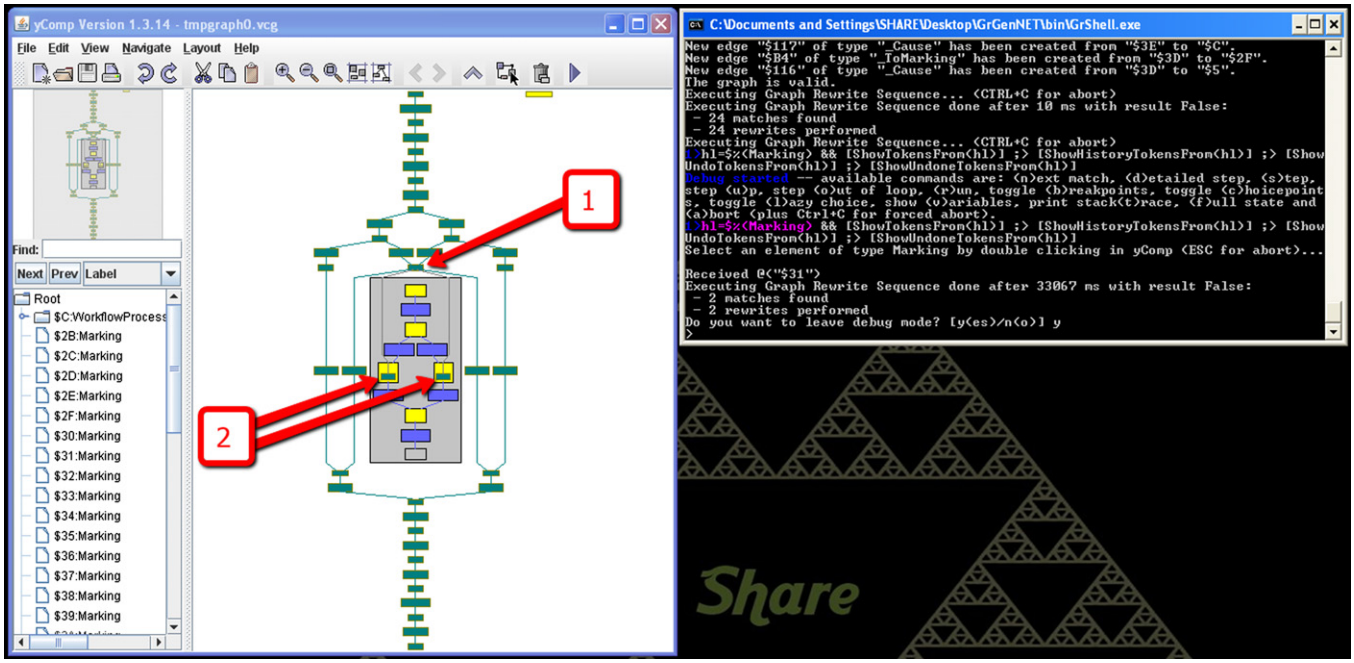


Fig. 52. Screenshot of the *Batch Execution* mode and the (optional) marking visualizer.

as well as lacking features. Although we have obviously regretted this, we want to emphasize that all issues were resolved within some weeks after reporting them to the GrGen.NET mailing list.

Opposed to these advantages, we identify the following points for improvement:

Statespace generation support. The primary source of over-technical details in our GrGen.NET implementation is the tool's lack of declarative statespace generation support. For example, Fig. 53 shows on lines 1 and 12 that rule *leaveTaskMoreOut* returns three parameters: (1) a parameter representing a copy of the original marking, (2) the updated marking, and (3) the BPMN element that should be shown to the user for characterizing a match of this rule. Also, on lines 8 and 9, the rule explicitly creates a copy of the current marking. Finally, the script that orchestrates the transformation rules for realizing scenarios two and three from Section 4.1 (i.e., scenario *Batch Statespace Generation* and *Interactive Statespace Generation*) contains rather complicated code. On top of the effort to write that code, there is obviously the effort to maintain it. Worst of all, we have learned late that the performance of the hand-written statespace generator is unacceptably poor and this can only be improved by resorting to the GrGen.NET C# API. Since alternative graph transformation tools such as Henshin [2] and GROOVE [47] provide declarative support for statespace generation, we have not invested effort in the C# workaround.

Documentation support. GrGen.NET is based on a textual language for encoding graph transformation rules. Although this is adequate during programming, a visual representation seems more appropriate for documentation purposes. There are visual graph transformation languages (see [49] for an overview) but most of these manipulate graphs in their abstract syntax representation. To the best of our knowledge, only AToM^3 [33] enables the specification of transformation rules in their concrete syntax form (e.g., directly in BPMN notation). Unfortunately, AToM^3 does not provide mechanisms for orchestrating and composing rules. Therefore, to date there is no plaform that can directly execute our visual rules.

```
1 rule leaveTaskMoreOut:(Marking,Marking,BaseElement) {
2   pi:ProcessInstance; cm:Marking; this:Task; tok:Token;
3   :HoldsThisToken(tok,this,cm,pi);
4   e:EnterSFandsplit(this,pi);
5   modify{
6     mNew:Marking;
7     exec(
8       rInitNewMarking(cm, mNew, pi) ;>
9       PIbackupPost ;>
10      leaveParallelGateway_RHS(this,tok,pi,cm)
11    );
12    return (mNew,cm,this);
13  }
14 }
15 pattern HoldsThisToken(tok:AbstrToken,fe:FlowElement, m:Marking, pi:
       ProcessInstance) {
16  independent {
17    pi -:itokens-> tok <-:Tokens- fe;
18    m -:Melem-> tok;
19  }
20 }
21 rule leaveParallelGateway_RHS(this:Activity,tok:Token,pi:ProcessInstance,cm:
       Marking) {
22  modify {
23    exec(rDelete(tok) ;> EnterSFandsplit_RHS(this,cm,pi));
24  }
25 }
26 pattern EnterSFandsplit(this:Activity, pi:ProcessInstance) modify (cm:Marking
       ) {
27   independent {
28    :HasMultipleNullOutFlows(this);
29  }
30  modify{
31    exec(EnterSFandsplit_RHS(this,cm,pi));
32  }
33 }
34
35 pattern HasMultipleNullOutFlows(this:Activity) {
36  <-sf1:SequenceFlow- this -sf2:SequenceFlow->;
37  if { sf1.Type=="NULL"&& sf2.Type=="NULL";}
38 }
39
40 rule EnterSFandsplit_RHS(this:Activity, cm:Marking, pi:ProcessInstance) {
41  iterated {
42    this -:_From-> sf:_SequenceFlow;
43    if {sf.Type=="NULL";}
44    modify {
45      cm -:Melem->  tok:Token <-:Tokens- sf;
46      pi -:itokens-> tok;
47    }
48  }
49  modify {}
50 }
```

Fig. 53. Code fragment of the implementation, based on GrGen.NET syntax.

The interested reader is encouraged to investigate the following seminal works to understand better the different focus of past research on graph-based visual language definition: the aforementioned Henshin tool is the successor of the Eclipse-oriented EMF Tiger tool [4], which is in turn the successor of the GenGED [3] tool. Related work from another group has produced DiaMeta, a tool which exposes DiaGen concepts to the Eclipse community [37,36]. The transformation definition style for all GenGED and DiaGen based tools is still at the abstract syntax level (similar to GrGen.NET). Their supportive transformation languages however do not yet support the advanced rule composition mechanisms that we have leveraged to keep our rule specifications concise. Instead, research has focussed mainly on layouting techniques. As an extension of our call to arms from Section 2.4, we invite the GenGED and DiaGen teams to action: they should support the specification (or generative documentation) of transformation rules in concrete syntax form to prevent inconsistencies between rule documentation and implementation.

It should be made possible to combine the strengths of transformation tools and visual language definition tools: we suggest that tools such as GrGen.NET provide a standard (e.g., XMI-based) representation of their advanced transformation programs. DiaGen and GenGED-based research and development could then focus on providing the proposed model-driven transformation documentation support.

4.3. Traceability, mental mapping to standard

We claim that the formalization of the BPMN 2.0 execution semantics in terms of graph transformation rules has a good traceability to the BPMN 2.0 standard. In that way, for each informal rule in the execution semantics, the corresponding formal graph transformation rule can easily be found and the correctness of the graph transformation rules can easily be checked. To an extent this traceability also applies to the implementation of the transformation rules in GrGen.NET.

To illustrate the traceability of the execution semantics rules to the GrGen.NET implementation of those rules, consider the following illustrative excerpt from the execution semantics [41]:

> *"An Activity MAY be a source for Sequence Flows; it can have multiple outgoing Sequence Flows. If there are multiple outgoing Sequence Flows, then this means that a separate parallel path is being created for each Sequence Flow (i.e., tokens will be generated for each outgoing Sequence Flow from the Activity)."*

This semantics is formalized by rule *leaveTaskMoreOut*, which is discussed in the context of Fig. 15. The traceability between the informal text and the visual rule representation is clear: the two sequence flows in the left-hand side of the rule clearly corresponds to the text fragment *"If there are multiple outgoing Sequence Flows"*, whereas the *iterated* block in the right-hand side of the rule clearly corresponds to the text fragment *"tokens will be generated for each outgoing Sequence Flow from the Activity"*.

Remark that it is impossible to have a complete correspondence between the informal text and the formal rewriting rule, since the informal text is often incomplete (e.g., the text fragment does not mention explicitly that a token should be removed from the activity). In fact, a complete correspondence is probably not even desirable, since the text is at a different abstraction level (it is not written with executability in mind). The visual representations from this paper aim to balance between these two representations. In summary, while direct correspondence is impossible and undesirable, the visual rule representations facilitate a good traceabiliy between our concise and executable implementation and the informal text from the standard.

Fig. 53 shows a fragment of the GrGen.NET based implementation. The fragment contains the implementation of rule *leaveTaskMoreOut* (cfr., Fig. 15) as well as various helper rules. Notice that the implementation of the rule takes just 14 (spaciously formatted) lines of code, the other code serves other rules too. The fragment also shows various nodes of type *ProcessInstance* and *Marking*. As stated in Section 3.2, such technical nodes are hidden in the visual rules to improve the documentation quality of these rules. More specifically, since all rules realize a transition from one *Marking* to another one, the rules in Section 3 never represent these two *Marking* nodes explicitly. The fragment also contains calls to rules *rInitNewMarking* and *PIbackupPost* that are not mentioned in Section 3. These helper rules realize some low-level plumbing for generating a statespace of the BPMN model. The fragment also contains some *independent* clauses (cfr., lines 16 and 27). These clauses ensure that the variables that are bound in the subpattern do not have to be isomorphic to previously bound variables. This pattern property is also mentioned in the text related to Fig. 15 but not shown explicitly on the figure.

In summary, the GrGen.NET code clearly contains more details than the visual rule representations from Section 3. Also, the code is more technical, since it is designed for maximal reuse across rules. Strong points of the implementation are that (1) it is also rule-based, (2) it leverages the same matching and control constructs as the visual rules and (3) all identifiers trace back to the conceptual discussion from Section 3.

4.4. Alternatives to the GrGen.NET implementation

As a summary of the previous sections, GrGen.NET has been an adequate platform for the implementation of this work. The debugging and performance features make the language and tool more suitable than alternatives. Considering the disadvantages, the lack of declarative statespace generation support has our current priority. More specifically, we have built a small Henshin based prototype as a second implementation of the rules from this paper. Besides Henshin's built-in support for statespace generation, the tool provides integration with popular Eclipse technologies. We have suspended further development of the Henshin prototype due to its currently too inconvenient approach to rule composition.

It should be emphasized that the contribution of this paper is not specific to the domain of graph transformation languages and tools. Clearly, the visual rule diagrams from Section 3 abstract from technical details, which benefits their applicability. One can, for example, also implement these rules in a general purpose programming language such as Java. This is important since most existing BPMN suites are *not* based on graph transformation languages.

Then again, there are various graph transformation tools that enable the embedding of graph transformation programs within a general purpose programming language. A notorious example is the Fujaba tool [7], that supports the seemless integration of visual transformation rules with Java statements. Various other tools (such as MoTMoT [38], AGG [4] and Henshin [2]) have adopted this approach and each of these tools has particular strengths and limitations.

5. Evaluation

The claims of this paper are that, using graph transformation rules, we can define an BPMN 2.0 execution semantics that is more complete and more easy to relate to the standard than other formalizations.

We evaluate the relative completeness of the BPMN 2.0 execution semantics in the next section, where we compare the execution semantics rules that are formalized in this paper to the execution semantics rules that are formalized in other papers. A summary of the conclusions is shown in Table 3, which shows clear support for this claim.

We argue that BPMN expert validation of graph transformation rules is relatively easy because such rules can be documented in the original BPMN 2.0 syntax. This claim can be verified by observing that all rules from this paper are indeed defined using the BPMN 2.0 syntax. We also argue that our approach provides better traceability than related work. This claim can be verified by observing that our rules update a BPMN model in-place instead of mapping it to another model, in another formalism.

Besides facilitating conceptual validation of our rules, we guarantee static type correctness of all rules (facilitated by GrGen.NET). Moreover, we have verified the expected behavior of the rules by testing our reference implementation using a comprehensive test-suite. Each of the rules in the BPMN 2.0 specification has a number of cases and exceptions. Using black-box testing principles and the guidelines of Soltenborn and Engels [54], we have created unit tests accordingly. For example, the BPMN 2.0 semantics specification reveals three different cases in handling an exclusive decision: (1) the case in which one of the conditions on the outgoing arcs evaluates to true and the token continues along that arc, (2) the case in which none of the conditions on the outgoing arcs evaluate to true and there is a default flow, in which case the token continues along the default flow, and (3) the case in which none of the conditions on the outgoing arcs evaluate to true and there is no default flow, in which case an exception is thrown. Consequently, we create a test case for each of these cases. The test cases that we identify in this way are presented in Table 1.

The expected outcomes of the test cases are not shown in the table. The outcomes of test cases are defined in terms of token distribution sequences (i.e., marking sequences) and can be derived from the BPMN 2.0 specification. For each testcase, we have manually evaluated whether the markings in the generated statespaces conform to what is mandated by the standard and our companion website makes the statespaces conveniently available for replay [9]. Moreover, our testsuite can be executed again and even adapted via [59].

Four test cases are split-up further into multiple test cases. These are: the inclusive join gateway, the error event, the signal event and the compensation event. The exclusive join has separate test cases for: the situation in which one or more tokens can arrive at the inclusive join and enable the join once they do; the situation in which a token can leave the path that leads to the inclusive join (via an exclusive split), such that it is no longer upstream of the inclusive join and the inclusive join may be enabled by other tokens; the situation in which the inclusive join is on a cycle. The error event has a test case for the regular case, in which a boundary event catches an error that occurs inside (the subprocess that is invoked by) an activity. In addition there are explicit test cases to test the case in which an error catch event only responds to an error with a specific name and the case in which an error catch event responds to all errors. The compensation event has a test case for the regular case, in which a compensation event activates the compensation activity of another activity. In addition there are explicit test cases to test a compensation event that activates the compensation of a specific activity and a compensation event that activates the compensation of all activities in a given subprocess.

Table 1
Test cases of semantics rules.

Concept	Test case
Sequence flow	Activity with one incoming flow
	Activity with no incoming flow
	Activity with multiple incoming flows
	Activity with one outgoing flow
	Activity with multiple outgoing flows
	Activity with multiple conditional flows without default flow
	Activity with multiple conditional flows with default flow
Process instantiation	Single start event
	Multiple start events
	Event-based gateway
Process termination	Single end event
	Multiple end events
	Terminate end event
	Implicit termination (no end event)
Subprocess instantiation	Single start event
	Multiple start events
Subprocess termination	Single end event
	Multiple end events
	Terminate end event
	Implicit termination (no end event)
Loop activity	Test before activity
	Test after activity
Gateway	Parallel gateway (split and join)
	Data-based exclusive gateway (split and join)
	Inclusive gateway split
	Inclusive gateway join
	All incoming flows have a token or no token upstream
	Tokens that are not upstream are ignored
	Unless the path visits the gateway
Events	Boundary event
	Error event
	A boundary event catches an error from inside the activity
	A boundary event may only catch errors with a given name
	A boundary event may catch all errors
	Signal event
	Intermediate event throws signal, caught by intermediate event
	End event throws signal, caught by intermediate event
	Signals are only caught if they have the right name
	Signals may be lost
	Compensation event
	Compensation end event compensates an activity
	Compensation intermediate event compensates an activity
	Compensation intermediate event compensates all activities

Besides these testcases that have been designed for unit testing, we have verified our reference implementation on a suite of real-life process models for integration testing. Fig. 54 shows a test model from this category. The model is based on the real-life sales recision process shown in Fig. 46. The business purpose of the model from Fig. 46 was to demonstrate how a failed credit card charge could be handled by a rollback of the payment process. The business purpose of the extension shown in Fig. 54 is to demonstrate how online shops could handle late deliveries for just-in-time supplier orders. The idea is that when not receiving a supplier order after some time, the supplier order is canceled and a new order is sent to another supplier. If even that fallback supplier does not deliver in time, the complete process (and the embedded customer payment process) should be compensated and the second supplier order should be canceled by a FAX message.

The technical purpose of the extension is twofold: first of all, the example involves the potential compensation of two processes (a

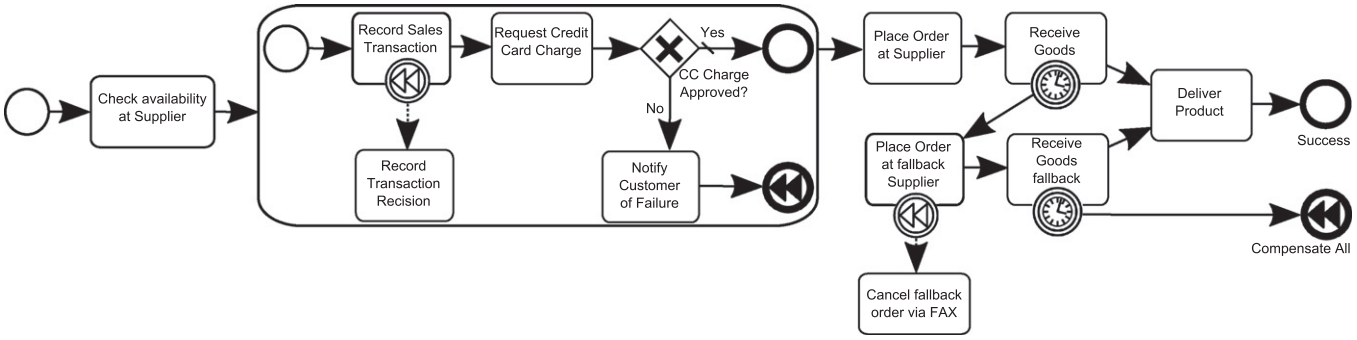


Fig. 54. Extension of the sales recision process, demonstrating a complex compensation.

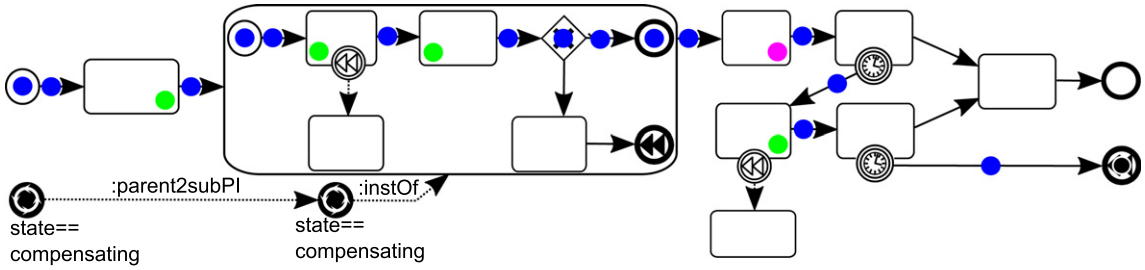


Fig. 55. After triggering rule *enterCompensationEndEvent* on the bottom-right element.

parent process and its subprocess). Secondly, the example demonstrates how the BPMN semantics prevents the double compensation of the same activity in the case of consecutive compensation throw events. More specifically, the "*Compensate All*" event at the top bottom of Fig. 54 should compensate all completed activities, *except* for "*Place Order at Supplier*" since that activity should already have been undone by the preceding "*Revoke order of late supplier*" compensation event and except for the two "*Receive Goods*" activities since these activities were interrupted by their associated boundary event. Note that the "*Compensate All*" event has no specific activity reference (which gives it the *compensate all* semantics) while the "*Revoke order of late supplier*" event refers specifically to the "*Place Order at Supplier*" activity (yielding local compensation semantics).

For this test model, our reference implementation generates a statespace of 50 marking nodes and 49 edges. The shape of the statespace graph is a tree with four leaves (not shown in a figure due to space considerations). Each path from root to leave represents one possible execution sequence. The one but longest execution sequence is noteworthy, since it demonstrates clearly that we have covered more challenging testcases than those considered by related work.

Fig. 55 shows an interesting marking on that sequence. The marking is the result after executing rule *enterCompensationEndEvent* on the compensation end event that is shown in the bottom-right of the figure. As intended, undo tokens are added to activities *Check availability at Supplier, Record Sales Transaction, Request Credit Card Charge, Place Order at Supplier* and *Place Order at fallback Supplier* since these activities have completed successfully (i.e., there was a history token in these activities). Activities *Receive Goods* and *Receive Goods fallback* do not receive an undo token since their tokens were deleted by rule *enterAutonomousBoundaryEvent* (instead of being retyped to a history token there). Both process instances are in state *compensating*.

After the marking shown in Fig. 55, rule *UndoProcessInstance* will apply. The marking is not shown due to space considerations but the reader is invited to reason about the effect of that rule: all undo tokens are retyped to undone tokens and the two compensation handling activities receive a regular token. After this marking, rule *completeSubprocessCompensation* leads to a final marking (i.e., a leaf in the statespace tree). In that marking, the two compensation handlers hold a history token and both process instances are in state *compensated*.

6. Related work

The BPMN 2.0 standard specifies the complete 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 [62,63,10,45,46,12,56]. These semantics differ with respect to the means that are used to specify the semantics, the goal with which the semantics is specified, the conceptual scope of the semantics and the BPMN constructs that are supported. Tables 2 and 3 provide an overview of these differences.

Wong and Gibbons [62,63] define a semantics for a subset of the BPMN control-flow concepts in terms of the process algebra CSP [50]. 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 reponse must be sent to the client within 24 h", and properties that apply to business process models in general, such as deadlock-freeness and proper completion [58]. We refer to the latter form of property checking as soundness checking. Dijkman et al. [10] 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 semantics precisely and to enable soundness checking. Prandi et al. [45] define a semantics in terms of a process algebra called COWS [44]. Their semantics allows for soundness checking and also of quantitative simulation of BPMN models, provided that

Table 2
Means, goals and conceptual scope of BPMN semantics.

Semantics	Means	Goals	Scope
BPMN 2.0 [41]	Natural language	Semantics specification	Complete
Wong and Gibbons [62,63]	CSP mapping	Property checking Soundness checking	Control-flow subset
Dijkman et al. [10]	Petri nets mapping	Semantics specification Soundness checking	Control-flow subset
Prandi et al. [45]	COWS mapping	Soundness checking Quantitative simulation	Control-flow subset Data-flow subset
Raedts et al. [46]	Petri nets mapping	Soundness checking	Control-flow subset
Dumas et al. [12]	Pseudo code	Semantics specification	OR-Join
Takemura [56]	Petri nets mapping	Semantics specification Soundness checking	Transactions
This paper	In-Place Graph Transformation rules	Semantics specification Ref. implementation	Control-flow subset

Table 3
Features supported by BPMN semantics.

Feature	BPMN standard [41]	Wong and Gibbons [62,63]	Dijkman et al. [10]	Prandi et al. [45]	Raedts et al. [46]	Dumas et al. [12]	Takemura [56]	This paper
Instantiation and termination								
Start event instantiation	X	X	X	X	X	X	X	X
Exclusive event-based gateway instantiation	X							X
Parallel event-based gateway instantiation	X							
Receive task instantiation	X							
Normal process completion	X	X	X	X	X	X	X	X
Activities								
Activity	X	X	X	X	X	X	X	X
Subprocess	X	X	X		X		X	X
Ad-hoc subprocesses	X							
Loop activity	X		X					X
Multiple instance activity	X							
Gateways								
Parallel gateway	X	X	X	X	X	X	X	X
Exclusive gateway	X	X	X	X	X	X	X	X
Inclusive gateway (split)	X	X	X	X		X		X
Inclusive gateway (merge)	X	X		X		X		X
Event-based gateway	X							
Complex gateway	X	X				X		
Events								
None events	X	X	X	X	X	X	X	X
Message events	X	X	X	X			X	X
Timer events	X	X						
Escalation events	X							
Error events (catch)	X	X	X	X			X	X
Error events (throw)	X		X	X				X
Cancel events	X	X						X
Compensation events	X						X	X
Conditional events	X							
Link events	X							X
Signal events	X							X
Multiple events	X							
Terminate events	X							X
Event subprocesses	X							

simulation information is provided with the model. The semantics is defined for a subset of both the control-flow and the data-flow aspect. Raedts et al. [46] 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. [12] 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 models that use this construct. Takemura [56] defines a precise semantics for the concepts that are related to BPMN transactions in terms of classical Petri nets. This formalization also supports soundness checking.

Tables 2 and 3 show that the formalization in this paper is unique with respect to its means, its prospective use and its com-

pleteness. All[5] existing formalizations map BPMN elements to more primitive counter-parts in other languages. For example, Wong and Gibbons map such elements to their CSP counter-parts [62,63] while Dijkman et al. [10] map them to Petri net elements. One drawback of such mapping based approaches is that in order to understand the BPMN semantics one first need to master another language too (CSP or Petri nets respectively). Another drawback of such approaches is that additional traceability techniques are needed which adds conceptual as well as technical complexity.

This paper uses in-place graph transformation rules to define the semantics. One benefit of using graph transformation rules is that a direct mapping is possible from the informal semantics specification to formal rewrite rules. Moreover, our rule representations are easily understandable since they are based on the standard visual representations of BPMN elements. Another benefit of using graph transformation rules is their 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 transformation systems. Table 3 supports this claim, by showing that the formalization of this paper is relatively complete and includes some notoriously hard concepts that are the sole focus of [12,56].

This paper is an extension of an earlier paper on the same topic [11]. The earlier paper is far less comprehensive than this one. In particular, the earlier paper does not cover any of the different event trigger types. Consequently, it also does not cover behavior regarding exception flow.

7. Conclusion

This paper proposes a formalization of the BPMN 2.0 execution semantics in terms of graph transformation rules. The paper shows that it is feasible to develop a complete formalization of the execution semantics in this way. It does that, both by explaining a large set of formal rules (covering also notoriously hard to formalize concepts such as the inclusive merge gateway and process compensation) and by indicating that this is the most complete BPMN 2.0 formalization to date.

The formalization is documented using visual rules that update a BPMN model in-place. All rules have been verified in a prototypical implementation based on the graph transformation platform GrGen.NET. There is good traceability from the informal execution semantics in the standard to the visual rules and their corresponding GrGen.NET implementation.

We show that the implementation can be used for various purposes, including interactive model execution as well as statespace generation with the purpose of doing (partial) statespace analyses. We claim that the implementation is particularly suited as a reference implementation of the execution semantics for the following reasons. The traceability to the informal execution semantics makes the graph transformation rules relatively easy to validate. The executability of the rules makes it possible to verify whether the effect of the rule set corresponds to what one expects conceptually. Moreover, the expressiveness makes it possible to do cover even the most complicated language elements.

As illustrated among others by Table 1, the implementation has already been tested thoroughly. Vendors can use our testsuite for checking semantic conformance of their tools. Regardless of rule representation syntax, our rule set is structured consistently (cfr., Figs. 10). This is particularly relevant for motivating vendors to align their implementations to our rules.

Based on our experience with GrGen.NET, we conclude that the benefits of this platform are that it is fast, it has a visual debugger and it is actively maintained. A drawback is that it lacks native support for specifying graph transformation rules directly in BPMN notation. Moreover, it does not provide built-in statespace generation support, such that specific graph transformation rules had to be written to generate BPMN statespaces. Consequently, alternatives to an implementation in GrGen.NET can be explored. It can even be envisioned to implement our visual rules directly in a programming language such as Java. We may invite corresponding experts to investigate the potential of alternative platforms via events such as the Transformation Tool Contest, noting that GrGen.NET has won various awards there already [61].

Acknowledgements

The authors thank Edgar Jakumeit for handling GrGen.NET feature requests and bug reports in a timely manner. They also wish to thank the anonymous reviewers for their valuable feedback. Moreover, Pieter Van Gorp wishes to thank Dirk Janssens and Reiko Heckel from the European Research and Training Network SegraVis (2-2001-00346) for sharing already in 2003 their inspiring ideas on graph transformation, visual modeling and semantics.

Appendix A. BPMN model definition

Throughout this paper, we rely upon a typed attributed graph (TAG)-based representation of BPMN models that is very close to the metamodel from the BPMN 2.0 standard from the Object Management Group (OMG [41]). The representation is slightly different since verbose structures have been simplified. Moreover, metamodel element names change frequently (and sometimes seemingly arbitrarily) in subsequent versions of OMG standards (cfr., [52]) and can be handled gracefully by specialized approaches [51,26]. Therefore, the terminology of our representation is not based on the very latest version of the OMG standard. Instead, it is based on terminology from a recent beta version thereof [40]. Graph-based metamodels can easily be related to OMG's metamodels [42,1]. Since some readers may however prefer an even more timeless mathematical formalization (e.g., to compare it to related work more easily), this appendix provides a clear mapping from mathematical sets and relations to the specific graph-oriented concepts from our reference implementation.

A BPMN 2.0 model is a tuple $(\mathcal{F}_e, \mathcal{F}_{name}, \mathcal{F}_e^{name}, \mathcal{F}_{cont}, \mathcal{F}_{cont'}^{el}, \mathcal{W}_{proc}, \mathcal{A}, \mathcal{A}_{ta}, \mathcal{A}_{ev}, \mathcal{E}_{name}, \mathcal{A}_{ev}^{\mathcal{E}_{name}}, \mathcal{E}_{code}, \mathcal{A}_{ev}^{\mathcal{E}_{code}}, \mathcal{A}_{sta}, \mathcal{A}_{end}, \mathcal{A}_{end}^{result}, \mathcal{A}_{im}, \mathcal{A}_{cat}, \mathcal{A}_{thr}, \mathcal{A}_{ev}^{trigger}, \mathcal{A}_{bl}, \mathcal{A}_{gw}, \mathcal{A}_{gw}^{type}, \mathcal{A}_{gw}^{excltype}, \mathcal{A}_{gw}^{inst}, \mathcal{L}_{std}, \mathcal{L}^{act}, \mathcal{S}_f, \mathcal{S}_f^{type}, \mathcal{A}_f, \mathcal{S}_f^{from}, \mathcal{S}_f^{to}, \mathcal{M}, \mathcal{M}_{el}, \rightarrow_{BPMN}, \mathcal{T}_{ok}, \mathcal{F}_{tok}, \mathcal{X}, \mathcal{P}_i, \mathcal{P}_i^{instOf}, \mathcal{P}_i^{child}, \mathcal{P}_i^{state}, \mathcal{P}_i^{mark}, \mathcal{P}_i^{tok}, \mathcal{P}_i^{exc}, \mathcal{T}_{ok}^{pi})$, where:

1. $\mathcal{F}_e$ is a finite set of flow elements,
   - $\mathcal{F}_{name}$ is a finite set of flow element names,
   - $\mathcal{F}_e^{name}$: $\mathcal{F}_e \rightarrow \mathcal{F}_{name}$ maps flow elements to their name,
2. $\mathcal{F}_{cont}$ is a finite set of flow element containers,
   - $\mathcal{F}_{cont}^{el}$: $\mathcal{F}_{cont} \times P(\mathcal{F}_e)$ (where $\mathcal{F}_{cont}^{el}{}^{-1}$ is a function) defines which flow elements belong to a container,
   - $\mathcal{F}_{cont'}^{el}$: $\mathcal{F}_{cont} \times \mathcal{F}_e = \{(x,y) | (x,z) \in \mathcal{F}_{cont}^{el} \wedge y \in z\}$ (for convenience),
3. $\mathcal{W}_{proc}$ is a finite set of workflow processes, with $\mathcal{W}_{proc} \subseteq \mathcal{F}_{cont}$,
4. $\mathcal{A}$ is a finite set of activities, with $\mathcal{A} \subseteq \mathcal{F}_e$,
   - $\mathcal{A}_{ta}$ is a finite set of tasks, with $\mathcal{A}_{ta} \subseteq \mathcal{A}$,
   - $\mathcal{A}_{ev}$ is a finite set of events, with $\mathcal{A}_{ev} \subseteq \mathcal{A}$,

[5] As an exception, Dumas et al. provide pseudo code performing in-place updates [12].

- $\mathcal{E}_{name}$ is a finite set of event definition names, used to match sender and receiver activities,
- $\mathcal{A}_{ev}^{\mathcal{E}_{name}}$: $\mathcal{A}_{ev} \rightarrow \mathcal{E}_{name}$ maps event activities to the name of the event under consideration,
- $\mathcal{E}_{code}$ is a finite set of error codes, used to match error signalers and handlers,
- $\mathcal{A}_{ev}^{\mathcal{E}_{code}}$: $\mathcal{A}_{ev} \rightarrow \mathcal{E}_{code}$ maps error event activities to error codes,

- $\mathcal{A}_{sta}$ is a finite set of start events, with $\mathcal{A}_{sta} \subseteq \mathcal{A}_{ev}$,
- $\mathcal{A}_{end}$ is a finite set of end events, with $\mathcal{A}_{end} \subseteq \mathcal{A}_{ev}$,
  - $\mathcal{A}_{end}^{result}$: $\mathcal{A}_{end} \rightarrow \mathcal{D}_{trigtype}$ defines the type of trigger that is generated by an end event,

- $\mathcal{A}_{im}$ is a finite set of intermediate events, with $\mathcal{A}_{im} \subseteq \mathcal{A}_{ev}$,
- $\mathcal{A}_{cat}$ is a finite set of intermediate catch events, with $\mathcal{A}_{cat} \subseteq \mathcal{A}_{im}$,
- $\mathcal{A}_{thr}$ is a finite set of intermediate throw events, with $\mathcal{A}_{thr} \subseteq \mathcal{A}_{im}$,
  - $\mathcal{A}_{ev}^{trigger}$: $(\mathcal{A}_{sta} \cup \mathcal{A}_{im}) \rightarrow \mathcal{D}_{trigtype}$ defines the type of trigger that an event is listening for,

- $\mathcal{A}_{bl}$ is a finite set of subprocess activities, with $\mathcal{A}_{bl} \subseteq (\mathcal{A} \cap \mathcal{F}_{cont})$,
- $\mathcal{A}_{gw}$ is a finite set of gateways, with $\mathcal{A}_{gw} \subseteq \mathcal{A}$,
  - $\mathcal{A}_{gw}^{type}$: $\mathcal{A}_{gw} \rightarrow \mathcal{D}_{gwtype}$ defines the type of a gateway element,
  - $\mathcal{A}_{gw}^{excltype}$: $\mathcal{A}_{gw} \rightarrow \mathcal{D}_{excltype}$ is a partial function that refines the type of exclusive gateway elements,
  - $\mathcal{A}_{gw}^{inst}$: $\mathcal{A}_{gw} \rightarrow \mathcal{D}_{bool}$ is a partial function that defines whether an event-based gateway can used to instantiate a process,

- $\mathcal{L}_{std}$ is a finite set of elements describing loop behavior,[6]
  - $\mathcal{T}_{before}$: $\mathcal{L}_{std} \rightarrow \mathcal{D}_{bool}$ is a function that define whether a loop test is executed before or after the iteration of the activity,

- $\mathcal{L}^{act}$: $\mathcal{A} \rightarrow \mathcal{L}_{std}$ defines the loop behavior (if any) of an activity,

5. $\mathcal{S}_f$ is a finite set of sequence flows, with $\mathcal{S}_f \subseteq \mathcal{F}_e$,
   - $\mathcal{S}_f^{type}$: $\mathcal{S}_f \rightarrow \mathcal{D}_{flowtype}$ defines the type of a sequence flow (conditional or catch-all),
6. $\mathcal{A}_f$ is a finite set of association flows, with $\mathcal{A}_f \subseteq \mathcal{F}_e$,
   - $\mathcal{S}_f^{from}$: $\mathcal{A} \times (\mathcal{S}_f \cup \mathcal{A}_f)$ (where $\mathcal{S}_f^{from^{-1}}$ is a function) defines the source of a sequence (or association) flow,
   - $\mathcal{S}_f^{to}$: $(\mathcal{S}_f \cup \mathcal{A}_f) \times \mathcal{A}$ defines the target of a sequence (or association) flow,
7. $\mathcal{M}$ is a finite set of markings,
   - $\mathcal{M}_{el}$: $\mathcal{M} \times \mathcal{T}_{ok}$ (where $\mathcal{M}_{el}^{-1}$ is a function) indicates which tokens belong to a specific marking,
   - $\rightarrow_{BPMN}$: $\mathcal{M} \times \mathcal{M}$ is the so-called transition relation of the BPMN model,
   - $\mathcal{T}_{ok}$ is a finite set of tokens,
   - $\mathcal{F}_{tok}$: $\mathcal{F}_e \times \mathcal{T}_{ok}$ (where $\mathcal{F}_{tok}^{-1}$ is a function) distributes tokens across the elements of a process model,
8. $\mathcal{X}$ is a finite set of exceptions,
9. $\mathcal{P}_i$ is a finite set of process instances,
   - $\mathcal{P}_i^{instOf}$: $\mathcal{P}_i \rightarrow \mathcal{F}_{cont}$ maps a process instance to its definition,
   - $\mathcal{P}_i^{child}$: $\mathcal{P}_i \times \mathcal{P}_i$ (where $\mathcal{P}_i^{child^{-1}}$ is a function) represents the parent/child relationship between process instances: a process instance is the parent of another process instance if one of its tokens has triggered the instantiation of the child,
   - $\mathcal{P}_i^{state}$: $\mathcal{P}_i \rightarrow \mathcal{D}_{pistate}$ maps a process instance to its state,
   - $\mathcal{P}_i^{mark}$: $\mathcal{P}_i \rightarrow \mathcal{M}$ binds a process instance to a specific marking,
   - $\mathcal{P}_i^{tok}$: $\mathcal{P}_i \times \mathcal{T}_{ok}$ (where $\mathcal{P}_i^{tok^{-1}}$ is a function) defines which tokens belong to a process instance,
   - $\mathcal{P}_i^{exc}$: $\mathcal{P}_i \times \mathcal{X}$ (where $\mathcal{P}_i^{exc^{-1}}$ is a function) indicates which exceptions were thrown by a specific process instance,
   - $\mathcal{T}_{ok}^{pi}$: $\mathcal{T}_{ok} \times \mathcal{P}_i$ (where $\mathcal{T}_{ok}^{pi^{-1}}$ is an injective function) captures which process instance (if any) is spawn from a token on a subprocess activity.

This definition relies on the enumerations $\mathcal{D}_{gwtype} = \{exclusive, inclusive, complex, parallel, event\}$, $\mathcal{D}_{excltype} = \{data, event\}$, $\mathcal{D}_{flowtype} = \{null, condition, otherwise\}$, $\mathcal{D}_{trigtype} = \{None, Message, Timer, Error, Cancel, Conditional, Link, Signal, Compensation, Multiple, Terminate\}$, $\mathcal{D}_{pistate} = \{active, terminated, failed, completing, completed, compensating, compensated\}$, and $\mathcal{D}_{bool} = \{true, false\}$.

In order to represent a BPMN Model as a Typed Attributed Graph, we define the following (attribute) node and (attribute) edge types.

- $T_{N_G} = \{FlowElement, FlowElementsContainer, WorkflowProcess, Activity, Task, Event, StartEvent, EndEvent, IntermediateEvent, IntermediateCatchEvent, IntermediateThrowEvent, BlockActivity, Gateway, LoopCharacteristics, SequenceFlow, FromTo, Association, Marking, Token, Exception, ProcessInstance\}$
- $T_{N_A} = \{String, TriggerType, GatewayType, ExclusiveType, SequenceFlowType, Boolean, PIstate\}$
- $T_{E_G} = \{Contains, LoopCharacteristicsOf, From, To, Melem, Mnext, Tokens, instOf, parent2subPI, itokens, ExceptionsOfPI, tok2pi\}$
- $T_{E_A} = \{eventDefinition, errorCode, Result, Trigger, TypeOfGW, TypeOfExcl, Instantiate, TestBefore, StateOf, Type\}$

Fig. 56, Fig. 57(a) and (b) show a graphical representation of these types. Elements from T_{N_G} are represented as classes, elements from T_{E_A} are represented as attributes, elements from T_{E_G} are represented as associations and elements from T_{N_A} are represented as enumerations.

A more formal treatment of type graphs is provided by Heckel et al. [25]. We have leveraged the GrGen.NET system to ensure that our sematic rules preserve conformance of instance graphs (models) to their type graph (metamodel).

The *type* function formalizes the mapping from mathematical BPMN 2 model elements to TAG types. A BPMN 2.0 graph is a Typed Attributed Graph representation $G_{BPMN} = (G, type)$ of a BPMN model $(\mathcal{F}_e, \mathcal{F}_{name}, \mathcal{F}_e^{name}, \mathcal{F}_{cont}, \mathcal{F}_{cont'}^{el}, \mathcal{W}_{proc}, \mathcal{A}, \mathcal{A}_{ta}, \mathcal{A}_{ev}, \mathcal{E}_{name}, \mathcal{A}_{ev}^{\mathcal{E}_{name}}, \mathcal{E}_{code}, \mathcal{A}_{ev}^{\mathcal{E}_{code}}, \mathcal{A}_{sta}, \mathcal{A}_{end}, \mathcal{A}_{end}^{result}, \mathcal{A}_{im}, \mathcal{A}_{cat}, \mathcal{A}_{thr}, \mathcal{A}_{ev}^{trigger}, \mathcal{A}_{bl}, \mathcal{A}_{gw}, \mathcal{A}_{gw}^{type}, \mathcal{A}_{gw}^{excltype}, \mathcal{A}_{gw}^{inst}, \mathcal{L}_{std}, \mathcal{L}^{act}, \mathcal{S}_f, \mathcal{S}_f^{type}, \mathcal{A}_f, \mathcal{S}_f^{from}, \mathcal{S}_f^{to}, \mathcal{M}, \mathcal{M}_{el}, \rightarrow_{BPMN}, \mathcal{T}_{ok}, \mathcal{F}_{tok}, \mathcal{X}, \mathcal{P}_i, \mathcal{P}_i^{instOf}, \mathcal{P}_i^{child}, \mathcal{P}_i^{state}, \mathcal{P}_i^{mark}, \mathcal{P}_i^{tok}, \mathcal{P}_i^{exc}, \mathcal{T}_{ok}^{pi})$, where the sets of elements that constitute the BPMN model are considered to be pairwise disjoint and:

- G is an attributed graph with nodes N_G, edges E_G, attribute nodes N_A and edges E_A that relate nodes to their attributes, where
  - $N_G = \mathcal{F}_e \cup \mathcal{W}_{proc} \cup \mathcal{L}_{std} \cup \mathcal{S}_f \cup \mathcal{A}_f \cup \mathcal{M} \cup \mathcal{T}_{ok} \cup \mathcal{X} \cup \mathcal{P}_i$
  - $N_A = \mathcal{F}_{name} \cup \mathcal{E}_{name} \cup \mathcal{E}_{code} \cup \mathcal{D}_{gwtype} \cup \mathcal{D}_{excltype} \cup \mathcal{D}_{trigtype} \cup \mathcal{D}_{pistate}$
  - $E_G = \mathcal{F}_{cont'}^{el} \cup \mathcal{L}^{act} \cup \mathcal{S}_f^{from} \cup \mathcal{S}_f^{to} \cup \mathcal{M}_{el} \cup (\rightarrow_{BPMN}) \cup \mathcal{F}_{tok} \cup \mathcal{P}_i^{instOf} \cup \mathcal{P}_i^{child} \cup \mathcal{P}_i^{mark} \cup \mathcal{P}_i^{tok} \cup \mathcal{P}_i^{exc} \cup \mathcal{T}_{ok}^{pi}$
  - $E_A = \mathcal{F}_e^{name} \cup \mathcal{A}_{ev}^{\mathcal{E}_{name}} \cup \mathcal{A}_{ev}^{\mathcal{E}_{code}} \cup \mathcal{A}_{gw}^{type} \cup \mathcal{A}_{gw}^{excltype} \cup \mathcal{P}_i^{state}$

- *type*, is the function that relates elements to their types, such that $type = \{(e, FlowElement) | e \in \mathcal{F}_e\} \cup \{(e, String) | e \in \mathcal{F}_{name}\} \cup \{(e, Name) | e \in \mathcal{F}_e^{name}\} \cup \{(e, FlowElementsContainer) | e \in \mathcal{F}_{cont} \cup \{(e, Contains) | e \in \mathcal{F}_{cont'}^{el}\} \cup \{(e, WorkflowProcess) | e \in \mathcal{W}_{proc}\} \cup$

[6] The loop test expression is outside the scope of this formalization.

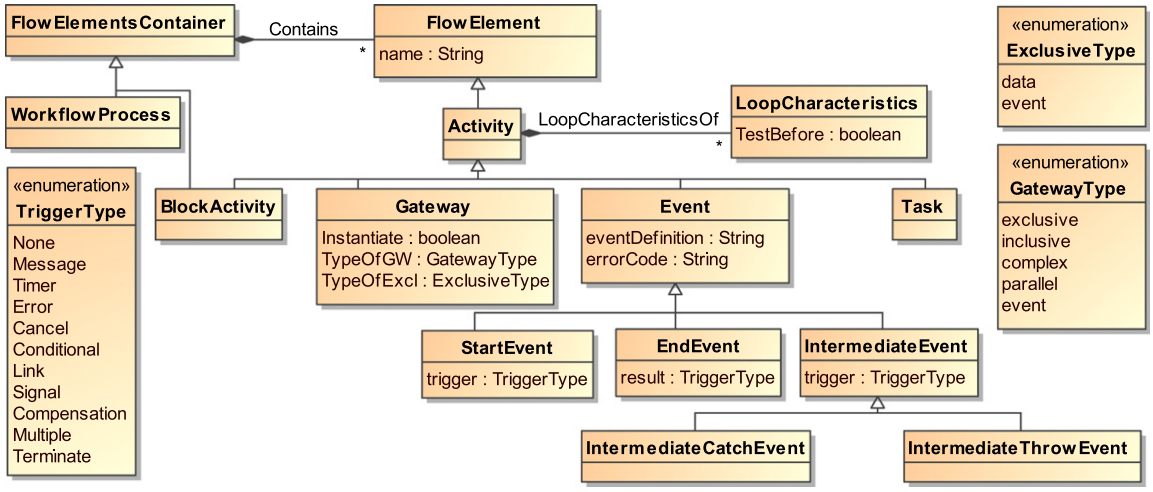
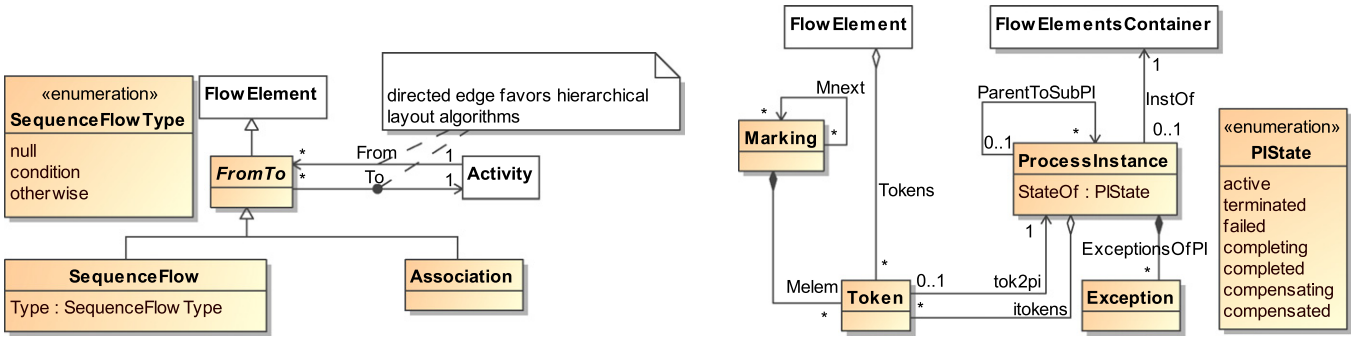


Fig. 56. BPMN types represented as a so-called *type* graph or *metamodel*.



(a) Types for sequence and compensation flows

(b) Types for process execution

Fig. 57. Classes without a background color are also defined in Fig. 57.

$\{(e, Activity)|e \in \mathcal{A}\} \cup \{(e, Task)|e \in \mathcal{A}_{ta}\} \cup \{(e, Event)|e \in \mathcal{A}_{ev}\}$
$\cup\{(e, String)|e \in \mathcal{E}_{name}\} \cup \{(e, EventDefinition)|e \in \mathcal{A}_{ev}^{\mathcal{E}_{name}}\} \cup$
$\{(e, String)|e \in \mathcal{E}_{code}\} \cup \{(e, errorCode)|e \in \mathcal{A}_{ev}^{\mathcal{E}_{code}}\} \cup \{(e, StartEvent)|e \in \mathcal{A}_{sta}\} \cup \{(e, EndEvent)|e \in \mathcal{A}_{end}\} \cup \{(e, Result)|e \in \mathcal{A}_{end}^{result}\} \cup \{(e, IntermediateEvent)|e \in \mathcal{A}_{im}\} \cup \{(e, Trigger)|e \in \mathcal{A}_{ev}^{trigger}\} \cup \{(e, IntermediateCatchEvent)|e \in \mathcal{A}_{cat}\} \cup \{(e, IntermediateThrowEvent)|e \in \mathcal{A}_{thr}\} \cup \{(e, BlockActivity)|e \in \mathcal{A}_{bl}\}$
$\cup\{(e, Gateway)|e \in \mathcal{A}_{gw}\} \cup \{(e, TypeOfGW)|e \in \mathcal{A}_{gw}^{type}\} \cup$
$\{(e, TypeOfExcl)|e \in \mathcal{A}_{gw}^{excltype}\} \cup \{(e, Instantiate)|e \in \mathcal{A}_{gw}^{inst}\} \cup$
$\{(e, LoopCharacteristics)|e \in \mathcal{L}_{std}\} \cup \{(e, TestBefore)|e \in \mathcal{T}_{before}\} \cup$
$\{(e, LoopCharacteristicsOf)|e \in \mathcal{L}^{act}\} \cup \{(e, SequenceFlow)|e \in \mathcal{S}_f\} \cup \{(e, Association)|e \in \mathcal{A}_f\} \cup \{(e, From)|e \in \mathcal{S}_f^{from}\} \cup$
$\{(e, To)|e \in \mathcal{S}_f^{to}\} \cup \{(e, Marking)|e \in \mathcal{M}\} \cup \{(e, Melem)|e \in \mathcal{M}_{el}\} \cup$
$\{(e, Mnext)|e \in \rightarrow_{BPMN}\} \cup \{(e, Token)|e \in \mathcal{T}_{ok}\} \cup \{(e, Tokens)|e \in \mathcal{F}_{tok}\} \cup \{(e, ProcessInstance)|e \in \mathcal{P}_i\} \cup \{(e, state)|e \in \mathcal{P}_i^{state}\} \cup$
$\{(e, instOf)|e \in \mathcal{P}_i^{instOf}\} \cup \{(e, parent2subPI)|e \in \mathcal{P}_i^{child}\} \cup \{(e, Exception)|e \in \mathcal{X}\} \cup \{(e, itokens)|e \in \mathcal{P}_i^{tok}\} \cup \{(e, tok2pi)|e \in \mathcal{T}_{ok}^{pi}\}$
$\cup\{(e, GatewayType)|e \in \mathcal{D}_{gwtype}\} \cup \{(e, ExclusiveType)|e \in \mathcal{D}_{excltype}\} \cup \{(e, Type)|e \in \mathcal{D}_{flowtype}\} \cup \{(e, TriggerType)|e \in \mathcal{D}_{trigtype}\}$
$\cup\{(e, PIState)|e \in \mathcal{D}_{pistate}\} \cup \{(e, Boolean)|e \in \mathcal{D}_{bool}\}$

Appendix B. BPMN model definition example

Using the definition from Appendix A, the example from Fig. 1 can be formalized as follows.

First, the different model elements must be defined: the tasks ($\mathcal{A}_{ta} = \{t_1, t_2\}$), the events ($\mathcal{A}_{sta} = \{s_1\}, \mathcal{A}_{end} = \{e_1\}, \mathcal{A}_{cat} = \{i_1\}$), the block activity ($\mathcal{A}_{bl} = \{bl_1\}$), the gateway ($\mathcal{A}_{gw} = \{gw_1\}$), the sequence flows ($\mathcal{S}_f = \{sf_1, sf_2, sf_3, sf_4, sf_5\}$) and the association flow that connects the block activity to its intermediate catch event ($\mathcal{A}_f = \{a_1\}$).

First, the different model elements must be defined: the tasks ($\mathcal{A}_{ta} = \{t_1, t_2\}$), the events ($\mathcal{A}_{sta} = \{s_1\}, \mathcal{A}_{end} = \{e_1\}, \mathcal{A}_{cat} = \{i_1\}$), the block activity ($\mathcal{A}_{bl} = \{bl_1\}$), the gateway ($\mathcal{A}_{gw} = \{gw_1\}$), the sequence flows ($\mathcal{S}_f = \{sf_1, sf_2, sf_3, sf_4, sf_5\}$) and the association flow that connects the block activity to its intermediate catch event ($\mathcal{A}_f = \{a_1\}$).

Second, we define the details of these elements. In particular, we define the names of the named tasks, event and block activity ($\mathcal{F}_e^{name} = \{(s_1, \text{'Order'}), (t_1, \text{'RegisterOrder'}), (t_2, \text{'ForwardtoWarehouse'}), (bl_1, \text{'ProcessOrder'})\}$); the relation between the block activity and its contents ($\mathcal{F}_{cont'}^{el} = \{(bl_1, t_1), (bl_1, t_2)\}$); the specifics of the gateway ($\mathcal{A}_{gw}^{type} = \{(gw_1, exclusive)\}, \mathcal{A}_{gw}^{excltype} = \{(gw_1, data)\}$); and the sequence flows and association flow that connect the different elements ($\mathcal{S}_f^{from} = \{(s_1, sf_1), (t_1, sf_2), (bl_1, sf_3), (gw_1, sf_4), (i_1, sf_5), (bl_1, a_1)\}, \mathcal{S}_f^{to} = \{(sf_1, bl_1), (sf_2, t_2), (sf_3, gw_1), (sf_4, e_1), (sf_5, gw_1)\}$ and $\mathcal{A}_f = \{(a_1, i_1)\}$).

We map these conceptual sets to the sets defining an attributed graph: N_G is defined as $\{t_1, t_2, bl_1, s_1, e_1, i_1, gw_1\}$ and N_A as {"*Order*", "*Register Order*", "*Forward to Warehouse*", "*Process Order*", *exclusive*, *data*}. As example graph edges (elements of E_G), consider the sequence flows (as defined in the previous paragraph). As example attribute edges (elements of E_A), consider the elements of $\mathcal{F}_e^{name}$ (also defined in the previous paragraph).

Finally, the *type* function is defined in such a way that each element is mapped to its most concrete type. For example, (t_1, *Task*) is an element of the *type* function of this BPMN 2.0 graph while the

fact that t_1 is conceptually also a *FlowElement* is not encoded in the *type* function. Instead, such inferences are made by inferences on the type graph (cfr., Fig. 56).

References

[1] C. Amelunxen, A. Königs, T. Rötschke, A. Schürr, MOFLON: a standard-compliant metamodeling framework with graph transformations, in: A. Rensink, J. Warmer (Eds.), Model Driven Architecture – Foundations and Applications: Second European Conference, Lecture Notes in Computer Science (LNCS), vol. 4066, Springer Verlag, Heidelberg, 2006, pp. 361–375.

[2] Thorsten Arendt, Enrico Biermann, Stefan Jurack, Christian Krause, Gabriele Taentzer, Henshin: advanced concepts and tools for in-place EMF model transformations, in: Dorina C. Petriu, Nic olas Rou quette, Øystein Haugen (Eds.), MoDELS, Lecture Notes in Computer Science, vol. 63, Springer, 2010, pp. 121–135.

[3] Roswitha Bardohl, Claudia Ermel, Ingo Weinhold, GenGED – a visual definition tool for visual modeling environments, in: John L. Pfaltz, Manfred Nagl, Boris Bhlen (Eds.), Proc. 2nd Intl. Workshop AGTIVE'03: Applications of Graph Transformation with Industrial Relevance, Charlottesville, USA, 2003, Revised and Invited Papers, Lecture Notes in Computer Science, vol. 3062, Springer, 2004, pp. 413–419.

[4] Enrico Biermann, Claudia Ermel, Leen Lambers, Ulrike Prange, Olga Runge, Gabriele Taentzer, Introduction to agg and emf tiger by modeling a conference scheduling system, STTT 12 (3–4) (2010) 245–261.

[5] Jakob Blomer, Rubino Geiß, Edgar Jakumeit, The GrGen.NET user manual, July 2011. http://www.info.uni-karlsruhe.de/software/grgen/GrGenNET-Manual.pdf.

[6] Sebastian Buchwald, Moritz Kroll, A GrGen.NET solution of the antworld case for the GraBaTs 2008 contest, in: Arend Rensink, Pieter Van Gorp (Eds), 4th International Workshop on Graph-Based Tools: The Contest, 2008.

[7] Sven Burmester, Holger Giese, Jög Niere, Matthias Tichy, Jörg P. Wadsack, Robert Wagner, Lothar Wendehals, Albert Zündorf, Tool integration at the meta-model level: the Fujaba approach, International Journal on Software Tools for Technology Transfer (STTT) 6 (2004) 203–218. 10.1007/s10009-004-0155-8.

[8] Juan de Lara, Roswitha Bardohl, Hartmut Ehrig, Karsten Ehrig, Ulrike Prange, Gabriele Taentzer, Attributed graph transformation with node type inheritance, Theoretical Computer Science 376 (May) (2007) 139–163.

[9] Remco Dijkman, Bpmn semantics companion webpage, December 2011. http://is.ieis.tue.nl/staff/rdijkman/bpmn.html.

[10] R.M. Dijkman, M. Dumas, C. Ouyang, Semantics and analysis of business process models in bpmn, Information and Software Technology (IST) 50 (12) (2008) 1281–1294.

[11] R.M. Dijkman, P. Van Gorp, Bpmn 20 execution semantics formalized as graph rewrite rules, in: Proceedings of the 2nd International Workshop on BPMN, Lecture Notes in Computer Science, vol. 67, Springer, 2010, pp. 16–30.

[12] Marlon Dumas, Alexander Grosskopf, Thomas Hettel, Moe Wynn, Semantics of standard process models with or-joins, in: Proceedings of OTM 2007, Part I, Lecture Notes in Computer Science, vol. 4803, 2007, pp. 41–58.

[13] Hartmut Ehrig, Karsten Ehrig, Ulrike Prange, Gabriele Taentzer, Fundamental theory for typed attributed graphs and graph transformation based on adhesive hlr categories, Fundamentae Informatica 74 (October) (2006) 31–61.

[14] Hartmut Ehrig, Reiko Heckel, Martin Korff, Michael Löwe, Leila Ribeiro, Annika Wagner, Andrea Corradini, Algebraic approaches to graph transformation – part ii: single pushout approach and comparison with double pushout approach, in: Grzegorz Rozenberg (Ed.), Handbook of Graph Grammars, World Scientific, 1997, pp. 247–312.

[15] Hartmut Ehrig, Michael Pfender, Hans Jürgen Schneider, Graph-grammars: an algebraic approach, in: FOCS, IEEE, 1973, pp. 167–180.

[16] Thorsten Fischer, Jörg Niere, Lars Torunski, Albert Zündorf, Story diagrams: a new graph rewrite language based on the unified modeling language and java, in: Hartmut Ehrig, Gregor Engels, Hans-Jörg Kreowski, Grzegorz Rozenberg (Eds.), Theory and Application of Graph Transformations, Lecture Notes in Computer Science, vol. 1764, Springer, Berlin/Heidelberg, 2000, pp. 157–167.

[17] Rubino Geiß, Graphersetzung mit Anwendungen im Übersetzerbau, PhD thesis, Universität Karlsruhe – Fakultät für Informatik, November 2008.

[18] Rubino Geiß, Gernot Veit Batz, Daniel Grund, Sebastian Hack, Adam M. Szalkowski, GrGen: A Fast SPO-Based Graph Rewriting Tool, 2006, pp. 383–397.

[19] Rubino Geiß, Moritz Kroll, On improvements of the varro benchmark for graph transformation tools, Technical Report 2007-7, Universität Karlsruhe, IPD Goos, December 2007, ISSN: 1432-7864.

[20] Kim Gostelow, Vincent G. Cerf, Gerald Estrin, Saul Volansky, Proper termination of flow-of-control in programs involving concurrent processes, SIGPLAN Not. 7 (November) (1972) 15–27.

[21] A. Habel, D. Plump, Computational completeness of programming languages based on graph transformation, in: Proc. FoSSaCS 2001, Lecture Notes in Computer Science, vol. 2030, 2001, pp. 230–245.

[22] Annegret Habel, Reiko Heckel, Gabriele Taentzer, Graph grammars with negative application conditions, Fundamenta Informaticae 26 (June) (1996) 287–313.

[23] Annegret Habel, Karl-Heinz Pennemann, Correctness of high-level transformation systems relative to nested conditions, Mathematical Structures in Computer Science 19 (2) (2009) 245–296.

[24] Reiko Heckel, Graph transformation in a nutshell, Electronic Notes in Theoretical Computer Science 148 (1) (2006) 187–198. Proceedings of the School of SegraVis Research Training Network on Foundations of Visual Modelling Techniques (FoVMT 2004).

[25] Reiko Heckel, Jochen Malte Küster, Gabriele Taentzer, Confluence of typed attributed graph transformation systems, in: Andrea Corradini, Hartmut Ehrig, Hans-Jörg Kreowski, Grzegorz Rozenberg (Eds.), IC GT, Lecture Notes in Computer Science, vol. 2505, Springer, 2002, pp. 161–176.

[26] Markus Herrmannsdoerfer, GMF: a model migration case for the transformation tool contest, in: Pieter Van Gorp, Steffen Mazanek, Louis Rose (Eds), TTC, EPTCS, vol. 74, 2011, pp. 1–5.

[27] Berthold Hoffmann, Edgar Jakumeit, Rubino Geiß, Graph rewrite rules with structural recursion, in: Graph Computation Models, 2008.

[28] Edgar Jakumeit, Mit GrGen.NET zu den sternen – erweiterung der regelsprache eines graphersetzungswerkzeugs um rekursive regeln mittels sterngraphgrammatiken und paargraphgrammatiken, Master's thesis, July 2008.

[29] Edgar Jakumeit, EBNF and SDT for GrGen.NET, in: Applications of Graph Transformation With Industrial Relevance, Budapest, Hungary, October 2011.

[30] Edgar Jakumeit, Sebastian Buchwald, Moritz Kroll, GrGen.NET – the expressive, convenient and fast graph rewrite system, International Journal on Software Tools for Technology Transfer (STTT) 12 (2010) 263–271, http://dx.doi.org/10.1007/s10009-010-0148-8.

[31] H.J. Kreowski, S. Kuske, A. Schürr, Nested graph transformation units, International Journal of Software Engineering and Knowledge Engineering 7 (4) (1997) 479–502.

[32] Leen Lambers, Hartmut Ehrig, Ulrike Prange, Fernando Orejas, Embedding and confluence of graph transformations with negative application conditions, in: Hartmut Ehrig, Reiko Heckel, Grzegorz Rozenberg, Gabriele Taentzer (Eds.), Graph Transformations, Lecture Notes in Computer Science, vol. 5214, Springer, Berlin/Heidelberg, 2008, pp. 162–177.

[33] Juan de Lara, Hans Vangheluwe, *AToM*^{3} A tool for multi-formalism and meta-modelling, in: Proceedings of the 5th International Conference on Fundamental Approaches to Software Engineering FASE '02, Springer-Verlag, London, UK, UK, 2002, pp. 174–188,.

[34] Tom Mens, Gabriele Taentzer, Olga Runge, Detecting structural refactoring conflicts using critical pair analysis, Electronic Notes in Theoretical Computer Science 127 (3) (2005) 113–128. Proceedings of the Workshop on Software Evolution through Transformations: Model-based vs. Implementation-level Solutions (SETra 2004).

[35] Tom Mens, Pieter Van Gorp, A taxonomy of model transformation, Electronic Notes Theoretical Computer Science 152 (March) (2006) 125–142.

[36] M. Minas, G. Viehstaedt, DiaGen: a generator for diagram editors providing direct manipulation and execution of diagrams, in: Proceedings of the 11th IEEE International Symposium on Visual Languages, September 1995, pp. 203–210.

[37] Mark Minas, Generating meta-model-based freehand editors, in: Albert Zündorf, Dániel Varró (Eds), Proc. of the 3rd International Workshop on Graph Based Tools (GraBaTs'06), Natal (Brazil), September 21–22, 2006, Satellite event of the 3rd International Conference on Graph Transformation, Electronic Communications of the EASST, vol. 1, 2006.

[38] Olaf Muliawan, Hans Schippers, Pieter Van Gorp, Model driven, Template based, Model Transformer (MoTMoT), 2007. http://motmot.sourceforge.net/.

[39] Tadao Murata, Petri nets: properties, analysis and applications, Proceedings of the IEEE 77 (4) (1989) 541–580.

[40] Object Management Group, Business process model and notation (BPMN) – v2.0, beta (dtc/2010-06-05), 2010. http://www.omg.org/spec/BPMN/.

[41] Object Management Group, Business process model and notation (BPMN) – v2.0 (formal/2011-01-03), 2011. http://www.omg.org/spec/BPMN/.

[42] Object Management Group, Meta Object Facility (MOF), 2011. http://www.omg.org/technology/documents/modeling_spec_catalog.htm.

[43] Detlef Plump, The graph programming language GP, in: Proc. Algebraic Informatics (CAI 2009), Springer-Verlag, 2009, pp. 99–122.

[44] D. Prandi, P. Quaglia, Stochastic COWS, in: Proceedings of ICSOC 2007, Lecture Notes in Computer Science, vol. 4749, 2007, pp. 245–256.

[45] Davide Prandi, Paola Quaglia, Nicola Zannone, Formal analysis of BPMN via a translation into COWS, in: Proc. COORDINATION 2008, Lecture Notes in Computer Science, vol. 5052, 2008, pp. 249–263.

[46] I.G.J. Raedts, M. Petkovic, Y.S. Usenko, J.M.E.M. van der Werf, J.F. Groote, L.J.A.M. Somers, 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, pp. 126–137.

[47] Arend Rensink, The GROOVE simulator: a tool for state space generation, in: Applications of Graph Transformations with Industrial Relevance (AGTIVE), Lecture Notes in Computer Science, vol. 3062, Springer, 2003, pp. 479–485.

[48] Arend Rensink, Nested quantification in graph transformation rules, in: Andrea Corradini, Hartmut Ehrig, Ugo Montanari, Leila Ribeiro, Grzegorz Rozenberg (Eds.), Graph Transformations, Lecture Notes in Computer Science, vol. 4172, Springer, Berlin/Heidelberg, 2006, pp. 1–13.

[49] Arend Rensink, Pieter Van Gorp, Graph transformation tool contest 2008, STTT 12 (3–4) (2010) 171–181.

[50] A.W. Roscoe, The Theory and Practice of Concurrency, Prentice-Hall, 1998.
[51] Louis Rose, Markus Herrmannsdoerfer, James Williams, Dimitrios Kolovos, Kelly Garcés, Richard Paige, Fiona Polack, A comparison of model migration tools, in: Dorina Petriu, Nicolas Rouquette, Oystein Haugen (Eds.), Model Driven Engineering Languages and Systems, Lecture Notes in Computer Science, vol. 6394, Springer, Berlin/Heidelberg, 2010, pp. 61–75. 10.1007/978-3-642-16145-2_5.
[52] Louis M. Rose, Dimitrios S. Kolovos, Richard F. Paige, Fiona A.C. Polack, Model migration case for TTC 2010, in: TTC'10: Transformation Tool Contest, 2010.
[53] Andy Schürr, Programmed graph transformations and graph transformation units in grace, in: Janice Cuny, Hartmut Ehrig, Gregor Engels, Grzegorz Rozenberg (Eds.), Graph Grammars and Their Application to Computer Science, Lecture Notes in Computer Science, vol. 1073, Springer, Berlin/Heidelberg, 1996, pp. 122–136.
[54] Christian Soltenborn, Gregor Engels, Towards test-driven semantics specification, in: Proceedings of the 12th International Conference on Model Driven Engineering Languages and Systems MODELS'09, Springer-Verlag, Berlin, Heidelberg, 2009, pp. 378–392.
[55] Christian Soltenborn, Gregor Engels, Using rule overriding to improve reusability and understandability of dynamic meta modeling specifications, Journal of Visual Languages and Computing 22 (3) (2011) 233–250. Special Issue on Visual Analytics and Visual Semantics.
[56] Tsukasa Takemura, Formal semantics and verification of BPMN transaction and compensation, in: Proceedings of the IEEE Asia-Pacific Conference on Services Computing, IEEE Computer Society, Los Alamitos, CA, USA, 2008, pp. 284–290.
[57] Wil M.P. van der Aalst, Workflow patterns, in: Ling Liu, M. Tamer Özsu (Eds.), Encyclopedia of Database Systems, Springer, US, 2009, pp. 3557–3558.
[58] W.M.P. van der Aalst, Verification of workflow nets, in: Proceedings of the 18th International Conference on Application and Theory of Petri Nets, 1997, pp. 407–426.
[59] Pieter Van Gorp, Online virtual machine related to this paper, December 2011. http://is.ieis.tue.nl/staff/pvgorp/share/?page=ConfigureNewSession&vdi=XP-TUe_BPMN_i_i.vdi.
[60] Pieter Van Gorp, Rik Eshuis, Transforming process models: executable rewrite rules versus a formalized java program, in: Dorina Petriu, Nicolas Rouquette, +ystein Haugen (Eds.), Model Driven Engineering Languages and Systems, Lecture Notes in Computer Science, vol. 6395, Springer, Berlin/Heidelberg, 2010, pp. 258–272.
[61] Pieter Van Gorp et al., Transformation tool contest 2011 – awards. http://is.ieis.tue.nl/staff/pvgorp/events/TTC2011/?page=Awards.
[62] Peter Y. Wong, Jeremy Gibbons, A process semantics for BPMN, in: Proceedings of the 10th International Conference on Formal Methods and Software Engineering, Lecture Notes in Computer Science, vol. 5256, 2008, pp. 355–374.
[63] Peter Y.H. Wong, Jeremy Gibbons, Formalisations and Applications of BPMN, In Science of Computer Programming 76 (2011) 633–650.