BPMN 2.0 execution semantics formalized as graph rewrite rules : extended version

Citation for published version (APA):
Van Gorp, P. M. E., & Dijkman, R. M. (2011). *BPMN 2.0 execution semantics formalized as graph rewrite rules : extended version*. (BETA publicatie : working papers; Vol. 353). Technische Universiteit Eindhoven.

Document status and date:
Published: 01/01/2011

Document Version:
Publisher's PDF, also known as Version of Record (includes final page, issue and volume numbers)

Please check the document version of this publication:

• A submitted manuscript is the version of the article upon submission and before peer-review. There can be important differences between the submitted version and the official published version of record. People interested in the research are advised to contact the author for the final version of the publication, or visit the DOI to the publisher's website.
• The final author version and the galley proof are versions of the publication after peer review.
• The final published version features the final layout of the paper including the volume, issue and page numbers.

Link to publication

General rights
Copyright and moral rights for the publications made accessible in the public portal are retained by the authors and/or other copyright owners and it is a condition of accessing publications that users recognise and abide by the legal requirements associated with these rights.

• Users may download and print one copy of any publication from the public portal for the purpose of private study or research.
• You may not further distribute the material or use it for any profit-making activity or commercial gain
• You may freely distribute the URL identifying the publication in the public portal.

If the publication is distributed under the terms of Article 25fa of the Dutch Copyright Act, indicated by the "Taverne" license above, please follow below link for the End User Agreement:

www.tue.nl/taverne

Take down policy
If you believe that this document breaches copyright please contact us at:

openaccess@tue.nl

providing details and we will investigate your claim.

Download date: 08. Feb. 2024

BPMN 2.0 Execution Semantics Formalized as Graph Rewrite Rules: extended version

Pieter van Gorp, Remco Dijkman

Beta Working Paper series 353

BETA publicatie	WP 353 (working paper)
ISBN	
ISSN	
NUR	982
Eindhoven	August 2011

BPMN 2.0 Execution Semantics Formalized as Graph Rewrite Rules: extended version

Pieter Van Gorp*, Remco Dijkman

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

Abstract

The Business Process Model and Notation (BPMN) standard version 2.0 informally defines a precise execution semantics. This paper defines that execution semantics formally, by defining the execution rules as graph rewrite rules. The paper shows that the formal definition of execution rules in this manner is intuitive and simple, in particular because they can be specified graphically, using the BPMN symbols, while maintaining mathematical rigour. Using graph rewriting tools, the resulting formal execution semantics can be used to directly execute models that are created in the BPMN. Therefore, it can be used as a reference implementation of the execution semantics and to test BPMN 2.0 engines, in combination with a set of BPMN test models that we also provide.

Key words: BPMN, Business Process Modeling, Formal Semantics

1. Introduction

The Business Process Model and Notation version 2.0 [20] 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, such as a workflow engine. The execution semantics is defined informally using natural language.

*Corresponding author (Email: p.m.e.v.gorp@tue.nl, Phone: +31-40-2472062)

Email addresses: `p.m.e.v.gorp@tue.nl` (Pieter Van Gorp), `r.m.dijkman@tue.nl` (Remco Dijkman)

There exist various initiatives to define a formal execution semantics in addition to the informal one [35, 36, 6, 23, 24, 7, 31]. 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 BPMN 2.0 specification itself.

This paper presents a formalization of the BPMN 2.0 execution semantics, using graph rewrite rules. Defining the execution semantics in this way has two important benefits.

Firstly, there is a direct traceability between the informal execution semantics rules in the BPMN 2.0 specification and their formal counterparts in a graph rewriting language. This facilitates easy verification of the correctness of each of the formal rules and simplifies their definition, as each execution semantics rule can be considered in isolation. The traceability between formal and informal rules exists, because:

1. each execution semantics rule can be represented separately as a graph rewrite rule, thus providing a one-to-one correspondence between informal and formal rules; and
2. the graph rewrite rules can be defined by using the BPMN 2.0 notation itself, thus showing a clear graphical relation between BPMN 2.0 notational elements and the formal execution semantics rules.

Secondly, using graph rewrite rules allows the semantics to be (relatively) complete. Theoretically, it is possible to develop a complete execution semantics of BPMN 2.0 in terms of graph rewrite rules, because graph rewrite rules are Turing complete [13]. This as opposed to, for example, classical Petri nets, in terms of which some constructs are notoriously hard to represent [6]. In addition to that, we show that our execution semantics covers more rules from the BPMN 2.0 specification than any other formal semantics so far.

In addition to that, there exist a wide variety of graph rewriting tools that can execute graph rewrite rules. This enables the rewrite rules defined in this paper to be executed in such a tool, making the execution semantics in terms of graph rewrite rules directly executable.

A formalization in terms of graph rewrite rules can be developed, because:

1. a BPMN 2.0 model can be interpreted as a typed, attributed graph on which the graph rewrite rules can be defined; and

2. the structure of a graph rewrite rule matches that of an execution semantics rule.

The second point holds, because the execution semantics is defined in terms of a token-game. It consists of rules that specify when a certain notational element can execute and what happens when it does. In this way, each rule in the execution semantics corresponds to a graph rewrite rule, which always have a “match” part and a “rewrite” part. The *match* part can be used to specify when a certain notational element can execute and the *rewrite* part can be used to specify what happens when it does.

The remainder of this paper is structured as follows. Section 2 provides an introduction to graph rewriting and BPMN 2.0 and it defines precisely how BPMN 2.0 process models can be interpreted as attributed graphs. Section 3 defines the BPMN 2.0 execution semantics formally, using graph rewrite rules. Section 4 explains how we implemented those rules in GrGen.NET. Section 5 presents related work on defining the BPMN 2.0 semantics formally and section 6 concludes.

2. Preliminaries

To define the BPMN 2.0 execution semantics in terms of graph rewrite rules, we must first show how a BPMN 2.0 model can be interpreted as a typed attributed graph, because graph rewrite rules are defined in that context. To this end Subsection 2.1 presents an introduction to typed attributed graphs and graph rewriting, Subsection 2.2 presents an introduction to BPMN 2.0 and Subsection 2.3 shows how a BPMN 2.0 model can be interpreted as a typed, attributed graph.

2.1. Typed Attributed Graphs and Graph Rewriting

This paper applies the paradigm of attributed graph rewriting with node type inheritance [5]. This paradigm is based on theoretical foundations that emerged in the 1970ies [11]. The language constructs that we apply to formalize the BPMN operational semantics have been formalized using Category Theory. Therefore, we can rely safely on high level abstractions without introducing ambiguity. Complementary to our contribution, others are leveraging the theoretical foundations of graph rewriting to build analysis tools. Section 4 illustrates how our BPMN formalization can benefit specifically from such tools.

Definition 1 (Attributed Graph). *An Attributed Graph* $G = ((N_G, N_A), (E_G, E_A), (source_G, source_A), (target_G, target_A))$ *consists of the sets:*

- N*, which is the set of nodes that is partitioned into the set* N_G *of graph nodes and* N_A *of attribute nodes;*
- E*, which is the set of edges that is partitioned into the set* E_G *of graph edges and* E_A *of attribute edges;*

and functions:

- *source, which indicates the source of each edge and can be partitioned into* $source_G : E_G \to N_G$ *and* $source_A : E_A \to N_G$*; and*
- *target, which indicates the target of each edge and can be partitioned into* $target_G : E_G \to N_G$ *and* $target_A : E_A \to N_A$*.*

Our definition of attributed graphs is based on that of E-graphs [9]. E-graphs however also support edge attributes, which are not used by this paper and hence omitted for the sake of simplicity.

Definition 2 (Typed, Attributed Graph). *Let* T_N *and* T_E *be sets of node and edge types respectively. A Typed, Attributed Graph is a tuple* $(G, type)$ *with:*

- $G = ((N_G, N_A), (E_G, E_A), (source_G, source_A), (target_G, target_A))$ *an attributed graph,*
- $type : (N \to T_N) \cup (E \to T_E)$ *a function which assigns a type to each node and edge.*

A graph rewrite rule defines how a typed, attributed graph can be rewritten into another typed attributed graph as follows.

Definition 3 (Graph Rewrite Rule). *A graph rewrite rule* $p = (G_{LHS}, G_{RHS})$ *consist of two Typed, Attributed Graphs which are called the left-hand side (LHS) and right-hand side (RHS) of* p*. For the application of a graph rewrite rule to a host graph* G_{host} *the following simplified algorithm can be used:*

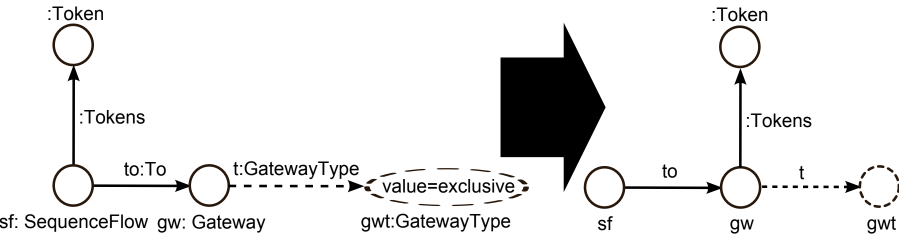


Figure 1: Graphical representation of a graph rewrite rule (flat graph syntax)

1. *Identify G_{LHS} within G_{host} . This involves finding a total graph morphism $m : G_{LHS} \rightarrow G_{host}$ that matches the left-hand side in the host graph. Unless specified otherwise, the morphism should be* isomorphic*, which means that each element from G_{LHS} should be mapped to at most one element in G_{host} (i.e., the mapping should be* injective.
2. *Delete all corresponding graph elements from G_{host}, w.r.t. m, that are part of G_{LHS} but not part of G_{RHS}.*
3. *Create a corresponding element in G_{host} for each element in G_{RHS} that is not in G_{LHS}.*
4. *Evaluate attribute updates that are defined on elements in p to their corresponding elements in G_{host}.*

Using this definition, graph rewrite 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, Figure 1 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 G_{LHS} should be mapped to an attribute node of type *GatewayType* with value exclusive from G_{host}.

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 sim-

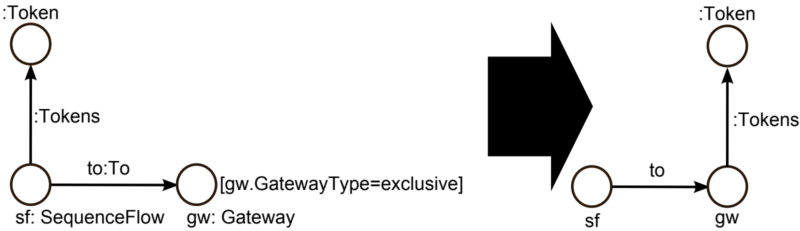


Figure 2: Graphical representation of a graph rewrite rule (attributed graph syntax)

plicity, attribute nodes and attribute edges are graphically embedded in their graph nodes in the remainder of this paper. Figure 2 follows this style to represent the same rule as that from figure 1.

There exist various extensions to the basic mechanisms of graph rewriting. Habel et al. [15] formalize so-called Negative Application Conditions (NACs). Such NACs enable the specification of G_{LHS} patterns that should *not* occur in G_{host} when checking the applicability of a rule. Interestingly, NACs were already introduced in 1996 [14] and they are of high practical relevance but only recent formal results have made analysis of rules with NACs feasible. We represent a NAC as a rounded rectangle around the part of the left-hand side that should *not* be part of the graph. We give this special type of rectangle a dashed border and annotate it with the label *NAC*. Figure 3 illustrates this. The figure shows a rule whose left-hand side consists of a particular graph node *gw* (a node of type *Gateway* with an attribute *GatewayType* set to *parallel*) that has *no* sequence flow pointing to it, on which there is *no* token. This double negation is formalized as a NAC that is nested within another NAC and also means that all incoming sequenceflows of *gw* should have at least one token. Remark that whenever NACs are not embedded in other NACs, one can use color (more specifically: *red*) to indicate which elements are in a negative pattern. In this paper, we once show how this shortcut notation can improve the readability of rules (cfr., Figures 15 and 16), but in general refrain from using this syntax, to avoid problems with grayscale printouts.

Most practical graph rewriting languages also include a set of control flow constructs [29, 16]. GrGen.NET provides a rule application control language with variables and logical as well as iterative control flow. This language can be used to control rules externally (e.g., for testing and debugging) as well as for internal control (to realize delegation). Other extensions of relevance to this paper are the arbitrary nesting of positive and negative patterns as well

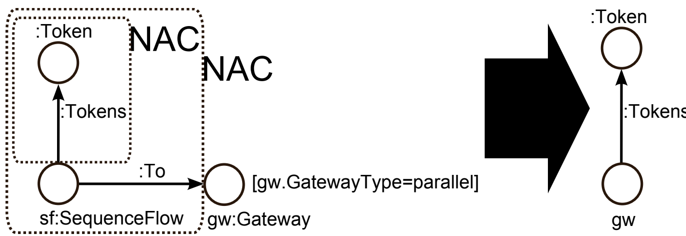


Figure 3: Graphical representation of a rule with a negative application condition

as subpatterns with a cardinality greater than one [26]. The latter extension involves G_{LHS} fragments that match to an a priori unknown number of subgraphs in G_{host}. In this paper, we represent such subpatterns again using a rounded rectangle with a dashed border and, unlike for NACs, we annotate it with the label *ITERATED* (cfr., Figure 10). Other language constructs are described upon first usage throughout this paper.

For a formal description of rule applications in terms of Category Theory, we refer to the handbook of graph grammars [10], where the foundations of the so-called single pushout (SPO) approach is reviewed.

2.2. 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.

Figure 4 shows a simple example of a BPMN model. 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 by the event of another message arriving. After either the subprocess completes or an order cancellation is received, the alternative paths are joined by an exclusive gateway, represented by the diamond with the "X". After that the process reaches the end event.

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*

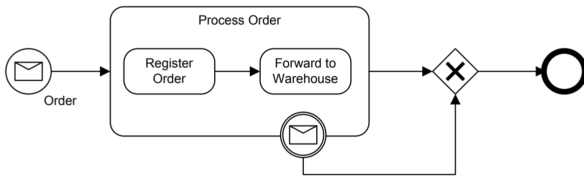


Figure 4: Example BPMN 2.0 model

at any incoming Sequence Flows activates the gateway". This is the same rule that is formalized by the graph rewrite rule in Figure 1 and 2.

The remaining BPMN 2.0 notational elements and execution semantics rules will be gradually introduced in section 3. It should be noted that BPMN 2 does not provide a standard expression language for specifying conditional guard expression. Although some workflow product vendors provide proprietary solutions to fill this gap, we decide not to formalize expressions. We believe this is quite appropriate, as BPMN is typically used for conceptual modeling. The proposed 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.

For precision, Appendix A presents an abstract syntax of the BPMN 2.0 notational elements in terms of mathematical sets. For example, it defines a set of activities and a set of sequence flows, such that each sequence flow has a relation to the activity from which it originates and a relation to the activity to which it points. According to this definition, a BPMN 2.0 model consists of an extension of the sets from the appendix. In the next subsection, we use this definition to interpret a BPMN 2.0 model as a typed attributed graph, which is necessary to define the graph rewrite rules.

The definition in the appendix includes two element types (i.e., $\mathcal{T}_{ok}$ and $\mathcal{M}$) that have no corresponding metaclass in the BPMN standard. The BPMN standard does define tokens as *"a theoretical concept that is used as an aid to define the behavior of a Process that is being performed"*, but excludes a token concept from the metamodel, since *"modeling and execution tools that implement BPMN are not required to implement any form of token"*. A marking is a related theoretical concept that denotes a distribution of tokens over the elements of a process model. A marking represents a concrete state of a running process model. Markings are especially relevant when analyzing process behavior. Throughout this article, we will also rely on an equivalence

relationship between markings. This enables the construction of a so-called *statespace* as an abstract representation of all possible executions of a process model.

2.3. Representing BPMN Models as Typed Attributed Graphs

Given the generic introduction to graph rewrite rules from Section 2.1 and the technology-independent abstract syntax definition of BPMN models from Section 2.2, this section proposes a graph-based representation of BPMN models. This enables us to define the BPMN 2.0 execution semantics rules in terms of graph rewrite rules in the next section.

The graph-based representation is constructed by defining:

1. the (attribute) node and (attribute) edge types that are used in the typed attributed graph;
2. which BPMN 2.0 notational elements become nodes in the types attributed graph, which become edges, which become attributes and which become attribute edges; and
3. which of the nodes, edges, attributes and attribute edges have which types.

The following (attribute) node and (attribute) edge types will be used in the typed attributed graph.

- T_{N_G} = {*FlowElement*, *FlowElementsContainer*, *WorkflowProcess*, *Activity*, *Task*, *Event*, *StartEvent*, *EndEvent*, *IntermediateEvent*, *IntermediateCatchEvent*, *IntermediateThrowEvent*, *BlockActivity*, *Gateway*, *LoopCharacteristics*, *SequenceFlow*, *Association*, *Marking*, *Token*, *Exception*, *ProcessInstance*}
- T_{N_A} = {*String*, *TriggerType*, *GatewayType*, *Boolean*, *PIstate*}
- T_{E_G} = {*Contains*, *LoopCharacteristicsOf*, *From*, *To*, *Melem*, *Mnext*, *Tokens*, *instOf*, *parent2subPI*, *pi2mark*, *itokens*, *ExceptionsOfPI*, *tok2pi*}
- T_{E_A} = {*EventDefinitionName*, *errorCode*, *Result*, *Trigger*, *TypeOfGW*, *TypeOfExcl*, *Instantiate*, *TestBefore*, *StateOf*}

Figures 5, 6(a) and 6(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.

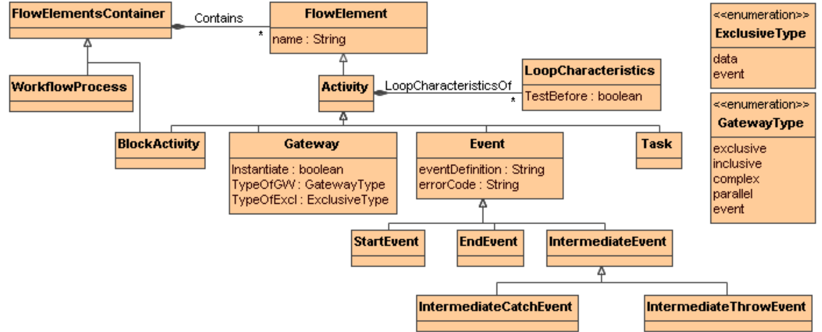
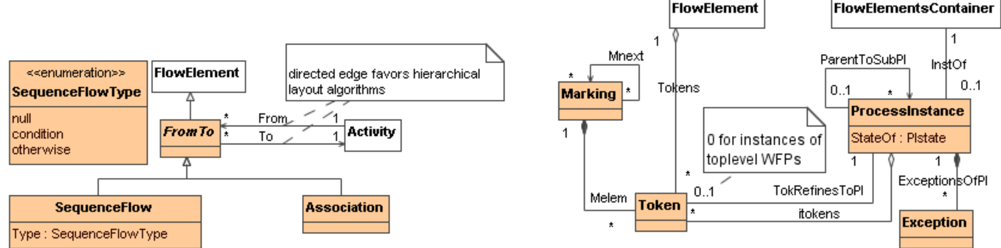


Figure 5: Type hierarchy of BPMN activity elements



(a) Types for sequence and compensation flows

(b) Types for process execution

Figure 6: Classes without a background color are also defined in figure 5.

We define the typed attributed graph by defining which of the elements from the BPMN 2.0 abstract syntax (as defined in Appendix A) become nodes, attributes, graph edges and attribute edges respectively. The *type* function formalizes the mapping from the sets in the abstract syntax from Appendix A to the types that are defined above. There are various options for defining this mapping. The mapping that we propose is optimized for pretty printing the graphs. The mapping of $\mathcal{S}_f$ elements to *nodes* may come as a surprize in this context: sequence flow elements should represent flow *links* between activities so it seems more natural to map them to graph *edges*. The reason for mapping them to nodes nevertheless, is that we need to be able to associate tokens with sequence flows too (as mandated by [20]) and our graph representation (as formalized in Section 2.1) does not support the linking of nodes and edges.

Consequently, 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:

- $G = ((N_G,\, N_A),\, (E_G,\, E_A),\, (source_G,\, source_A),\, (target_G,\, target_A))$,
  - 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= $\mathcal{F}_e^{name} \cup \mathcal{A}_{ev}^{\mathcal{E}_{name}} \cup \mathcal{A}_{ev}^{\mathcal{E}_{code}} \cup \mathcal{A}_{gw}^{type} \cup \mathcal{A}_{gw}^{excltype} \cup \mathcal{P}_i^{state}$
  - $source_G$: $\{((x_1, x_2), x_1) | (x_1, x_2) \in E_G\}$,
    $source_A$= $\{((x_1, x_2), x_1) | (x_1, x_2) \in E_A\}$
  - $target_G$= $\{((x_1, x_2), x_2) | (x_1, x_2) \in E_G\}$,
    $target_A$= $\{((x_1, x_2), x_2) | (x_1, x_2) \in E_A\}$

- type: $(N \rightarrow T_N) \cup (E \rightarrow T_E)$=
  { (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 $\in \mathcal{W}_{proc}$} $\cup$
  { (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, *EventDefinitionName*) | 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*) $\mid$ e $\in \mathcal{S}_f\} \cup \{$ (e, *Association*) $\mid$ e $\in \mathcal{A}_f\} \cup$
$\{$ (e, *From*) $\mid$ e $\in \mathcal{S}_f^{from}\} \cup \{$ (e, *To*) $\mid$ e $\in \mathcal{S}_f^{to}\} \cup$
$\{$ (e, *Marking*) $\mid$ e $\in \mathcal{M}\} \cup \{$ (e, *Melem*) $\mid$ e $\in \mathcal{M}_{el}\} \cup$
$\{$ (e, *Mnext*) $\mid$ e $\in \rightarrow_{BPMN}\} \cup \{$ (e, *Token*) $\mid$ e $\in \mathcal{T}_{ok}\} \cup$
$\{$ (e, *Tokens*) $\mid$ e $\in \mathcal{F}_{tok}\} \cup \{$ (e, *ProcessInstance*) $\mid$ e $\in \mathcal{P}_i\} \cup$
$\{$ (e, *state*) $\mid$ e $\in \mathcal{P}_i^{state}\} \cup \{$ (e, *instOf*) $\mid$ e $\in \mathcal{P}_i^{instOf}\} \cup$
$\{$ (e, *parent2subPI*) $\mid$ e $\in \mathcal{P}_i^{child}\} \cup \{$ (e, *Exception*) $\mid$ e $\in \mathcal{X}\} \cup$
$\{$ (e, *itokens*) $\mid$ e $\in \mathcal{P}_i^{tok}\} \cup \{$ (e, *pi2exc*) $\mid$ e $\in \mathcal{P}_i^{exc}\} \cup$
$\{$ (e, *tok2pi*) $\mid$ e $\in \mathcal{T}_{ok}^{pi}\} \cup \{$ (e, *GatewayType*) $\mid$ e $\in \mathcal{D}_{gwtype}\} \cup$
$\{$ (e, *ExclusiveType*) $\mid$ e $\in \mathcal{D}_{excltype}\} \cup \{$ (e, *Type*) $\mid$ e $\in \mathcal{D}_{flowtype}\} \cup$
$\{$ (e, *TriggerType*) $\mid$ e $\in \mathcal{D}_{trigtype}\} \cup$
$\{$ (e, *ProcessInstanceState*) $\mid$ e $\in \mathcal{D}_{pistate}\} \cup \{$ (e, *Boolean*) $\mid$ e $\in \mathcal{D}_{bool}\}$

Although a direct visualization of this graph-based structure resembles to a large extent a BPMN 2.0 model in the BPMN 2.0 notation, two types of annotations need to be defined. Firstly, some elements (edges and nodes) from a BPMN model are not visible on BPMN diagrams. Secondly, some edges are visualized by visually embedding the target node within the source node (typically if the edge types are marked as compositions in the type graph). Finally, some elements are pretty printed with a particular icon, based on their type.

More specifically, nodes of type *ProcessInstance* (and consequently the related edges) are not shown on standard BPMN diagrams. Similarly, since tokens are not formalized in the BPMN standard, they have no *standard* visual representation either. Tokens do occur in BPMN related languages, such as Petri-Nets, and in such cases they are visually embedded in their corresponding flow element. Therefore, we embed nodes of type *Token* in their corresponding *FlowElement* node (based on the edge of type *Tokens*). Also, instead of representing the *Contains* edge as an arc, the target nodes of such arcs are embedded within the source nodes.

This paper does not discuss which icons are associated with standard element types, since that is evident from the BPMN standard and related textbooks [20, 30]. It is worth mentioning however, that for readability purposes, we introduce a new icon for representing *ProcessInstance* nodes (cfr., *piNew* in Figure 7). Nodes of type *Token* are represented as a black dot (cfr., *tNew* in Figure 7), which is inspired by the Petri-Net syntax.

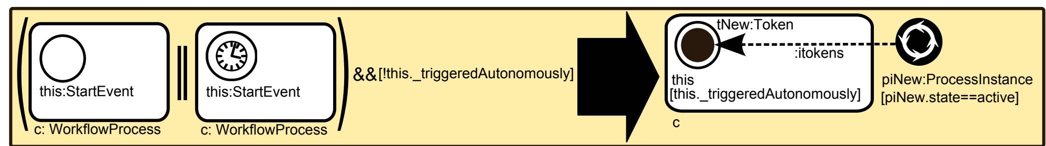


Figure 7: Autonomous Instantiation: rule *enterAutonomousStartEvent*.

3. Execution Semantics

This section defines the execution semantics of BPMN 2.0 in terms of graph rewrite rules. The section has the same structure as the BPMN 2.0 specification's [20] chapter on the execution semantics, to maintain good traceability between the execution semantics rules in the specification and the graph rewrite rules that formalize them.

Figure 8 and 9 provide 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 rewrite 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.

3.1. Process Instantiation and Termination

Autonomous Instantiation. Figure 7 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). Remark that the rule also checks and updates an attribute *_triggeredAutonomously* on the start event. This attribute should not be considered as part of what the BPMN standard prescribes. We have added it to ensure the termination of our rule set: the check/update ensures that this rule fires only once for each start event. This does not restrict our solution space, since we are only interested in analyzing all *different* execution scenario's. The check/update should be removed when using the proposed rule set for general process execution (e.g., in a production workflow engine).

Normal Termination. Figure 10 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 clauses in the left-hand side. The first clause of the rule's left-hand side states that the rule applies

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		enterParallelGateway leaveParallelGateway
Exclusive gateway		enterExclusiveGateway leaveDataExclusiveGateway catchImplicitlyThrownException
Inclusive gateway		enterInclusiveGateway leaveInclusiveGateway catchImplicitlyThrownException

Figure 8: Overview of BPMN 2.0 concepts with execution semantics rules (1/2)

Event Type	Notation	Rewrite Rule
None Start events		enterAutonomousStartEvent, enterSubProcess, leaveStartEvent
Intermediate events		enterIntermediateThrowEvent (See Compensation, Message, Signal)
Intermediate boundary events		enterAutonomousBoundaryEvent
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, UndoProcessInstance, completeProcessNormal leaveCompensationIntermediateThrowEvent
Signal events		enterIntermediateThrowEvent, enterEndEvent, leaveSignalThrowEvent, enterSignalCatchIntermediateEvent, enterSignalCatchStartEvent

Figure 9: overview of BPMN 2.0 concepts with execution semantics rules (2/2)

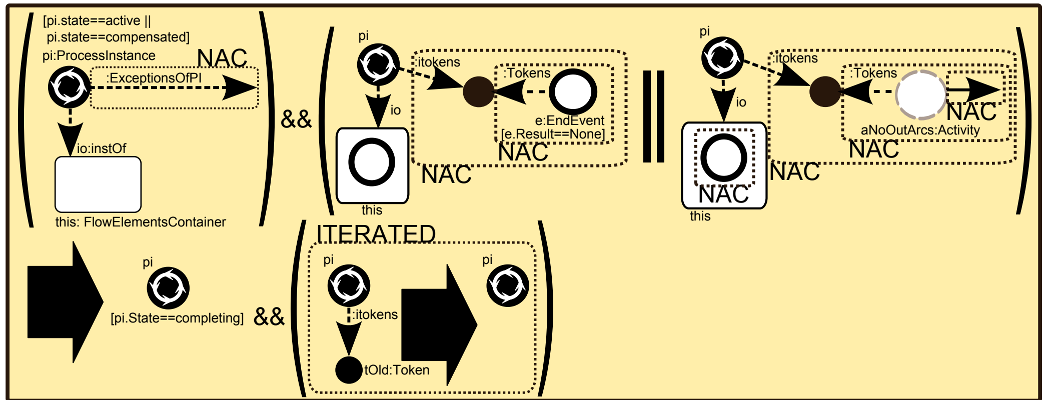


Figure 10: Normal termination of a process: rule *completeProcessNormal*.

to situations with a process instance that is in the *active* or *compensated*[1] state. The NAC in this clause states that no exceptions should have been thrown for this process instance. The second clause (i.e., the part 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 clause is satisfied too, the left-hand side matches and the right-hand side can be applied. The implementation of the right-hand side has been extracted to a separate rule – *completeProcess_RHS* – since its definition turns out to be useful elsewhere too.

3.2. Activities

Sequence Flow Considerations. Figure 11 shows the rule called *leaveTaskOneOut*. 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 (see the *Type* attribute of *SequenceFlow* nodes). 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 implicitly *homomorphic*:

[1]Details about compensation are discussed in the context of compensation events.

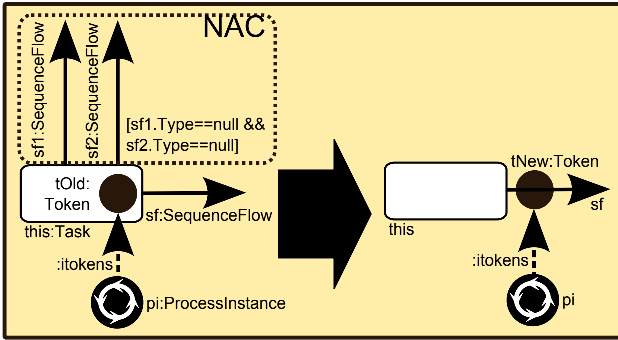


Figure 11: Regular sequence flow for task elements (1/2): rule *leaveTaskOneOut*.

the NAC expresses a pattern of two outgoing sequence flow (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.

Recall from Section 2.1 that sequence flows are in fact represented as *nodes* in BPMN 2.0 graphs. Therefore, the relation between the node *tNew* and the node *sf* is realized by a graph edge of type *Tokens* (cfr., Figure 6(b)). Nodes of type *SequenceFlow* are visualized as edges in the visual representations of our rewrite rules, to resemble the concrete syntax of BPMN 2.0 more closely. More specifically, Figure 11 is a complete visualization of all rule variables. However, since, the purpose of the figures in this paper is mainly to *document* the graph rewrite rules, we omit the *pi* node from both the left- and right-hand side in the figure, since it is quite trivial that throughout a process execution, the tokens remain within the same instance. All subsequent figures will abstract from *ProcessInstance* nodes, unless such nodes are updated by the rewrite rule. For example, for Figures 12 and 13, we omit the node and edges that ensure that *tOld* and *tNew* belong to the same process instance. In contrast, Figure 14 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.

Figure 12 shows rule *leaveTaskMoreOut*. This rule complements rule 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 [32], 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

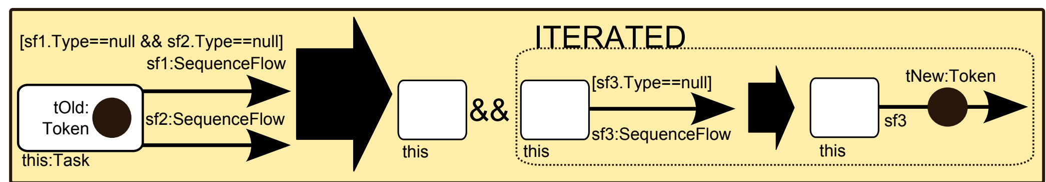


Figure 12: *AND split* sequence flow for task elements: rule *leaveTaskMoreOut*.

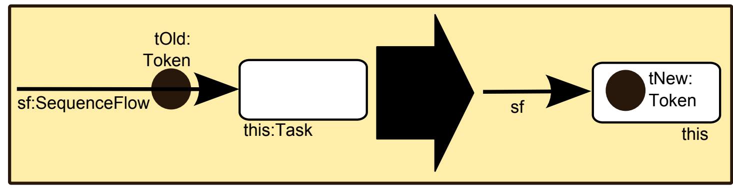


Figure 13: Regular sequence flow for task elements (2/2): rule *enterTask*.

outgoing sequence flows that are of type *null*. In the right-hand side, the token is removed from the task. Additionally, each outgoing sequence flow of type *null* gets a new token assigned to it. Obviously, all new tokens should go to the same process instance as the one that produced the match. Remark that variables *sf1* and *sf2* are allowed to be bound to the same element as variable *sf3*. In graph rewrite rule jargon, for these variables we want homomorphic matching rather than (the default) isomorphic matching. Put differently, we allow the left-hand side patterns of both subrules to be mapped *non-injectively* to nodes in the host graph.

As a symmetric counter-part of rule *leaveTaskOneOut*, consider rule *enterTask*, which is visualized in Figure 13. Remark that this rule also applies to the situation where the task node has multiple incoming sequence flows. If this is the case, extra sequence flows *without* a token do not prohibit the firing of rule *enterTask* (there is no NAC related to incoming flows besides *sf*). 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.

Sub-Process/Call Activity. Figure 14 shows the rule called *enterSubprocess*, which formalizes subprocess invocations. The left-hand side consists of a pattern with *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

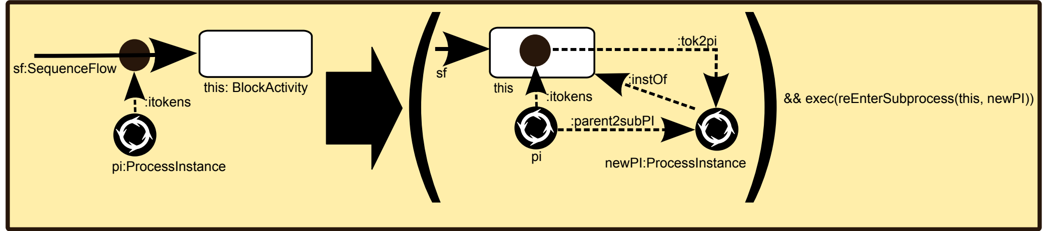


Figure 14: Instantiation of subprocesses: rule *enterSubprocess*.

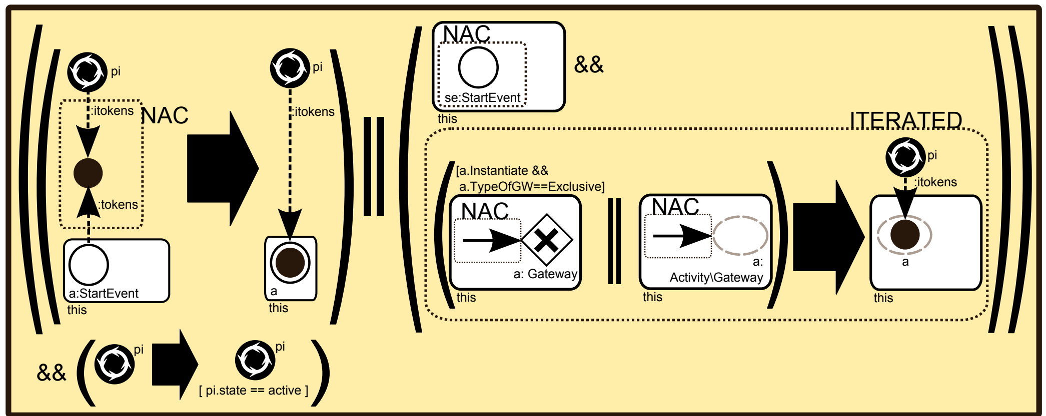


Figure 15: Helper *reEnterSubprocess(this:BlockActivity,pi: ProcessInstance)*.

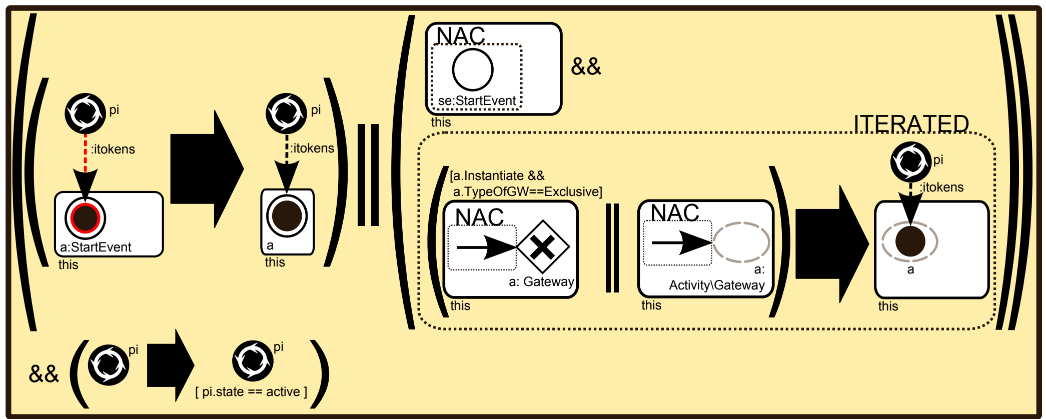


Figure 16: Helper *reEnterSubprocess*, with the color-based NAC syntax.

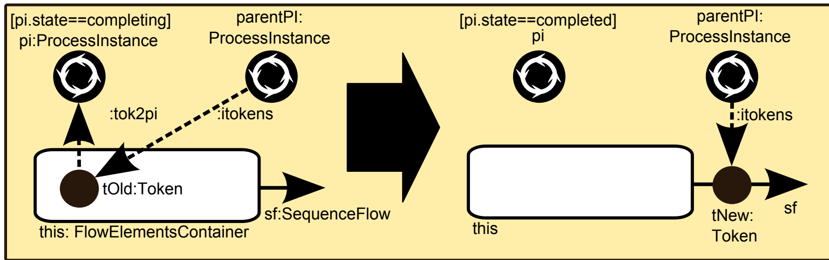


Figure 17: Leaving a block activity: rule *leaveSubprocessNormal*.

the process instance level. More specifically, in the right-hand side of the rule from Figure 14, a new process instance node *newPI* is created and relations to original process instance and the block activity are established. Finally, *enterSubprocess* calls rule *reEnterSubprocess* and passes the block activity (*this*) and process instance (*pi*) as arguments.

Rule *reEnterSubprocess* is shown on Figure 16. Remark that this figure relies on the color *red* to represent the elements that are in a NAC. Figure 14 shows the equivalent rule in the more verbose (yet grayscale printer friendly) conventional syntax. Since variables *this* and *pi* are passed as rule parameters, they are already bound. Therefore, they do not contain their expected type even in the left-hand side of the rule. Rule *reEnterSubprocess* consists of two subrules that are matched alternatively (i.e., they are combined with the "‖" operator).

On the left side of the "‖" operator on Figure 16, we see the first subrule. This subrule matches if the block 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*.

On the right side of the "‖" operator on Figure 16, we see the second subrule of *reEnterSubprocess*. This subrule handles the situation where block activity *this* does not have a start event. For such subprocesses, two types of elements without incoming arcs will 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).

Figure 17 shows the rule called *leaveSubprocessNormal*. The rule matches when a process instance is in the *completing* state. Remark that rule *completeProcessNormal* (shown on Figure 10) rewrites process instances to that

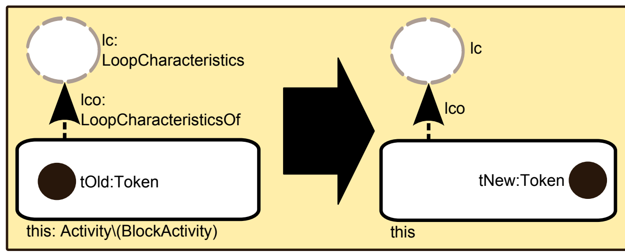


Figure 18: Iterating a loop for a regular activity: rule *reEnterLoopActivity*.

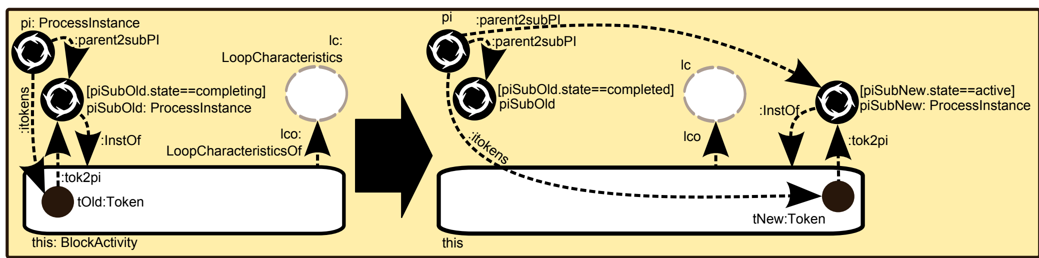


Figure 19: Iterating a loop for subprocess activities: rule *reEnterLoopSubprocess*.

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 an outgoing sequence flow.

Loop Activity. Figure 18 shows rule *reEnterLoopActivity*. This rule applies to all activity types but block activities (i.e., subprocesses). Since the left-hand side specifies a situation with a token on the activity, it expresses the situation where the loop body has just been executed. In this situation, both a *while-do* loop as a *repeat-until* loop can enter its loop body again. Therefore, there is no constraint on the *testBefore* attribute of the *lc* node.

Figure 19 shows the rule called *reEnterLoopSubprocess*. The rule realizes loop behavior for block activities. Remark that a new process instance, *piSubNew*, is spawned in the right-hand side. Also, the *state* attribute of the old and new instances of the subprocess are set properly: the old instance's state is updated from *completing* to *completed* whereas the new instance's state is initialized to *active*. Also note that the configuration of the edges with respective types *InstOf*, *tok2pi* and *parent2subPI* is rather complex. This emphasizes the importance of specifying the operational semantics formally.

Figure 20 shows an initial version of the rule for skipping a loop activity with *test before* characteristics (i.e., it is a rule for skipping a *do-while* loop).

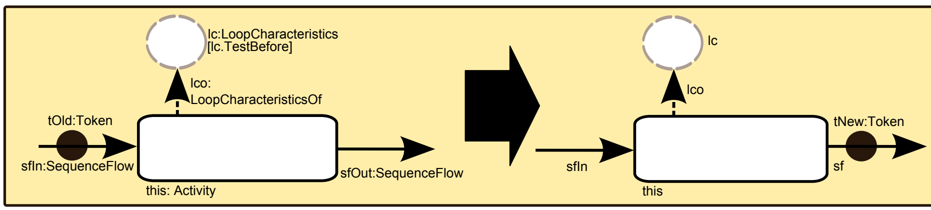


Figure 20: Skipping a *while-do* loop for regular *or subprocess* activities: *skipLoopActivity*.

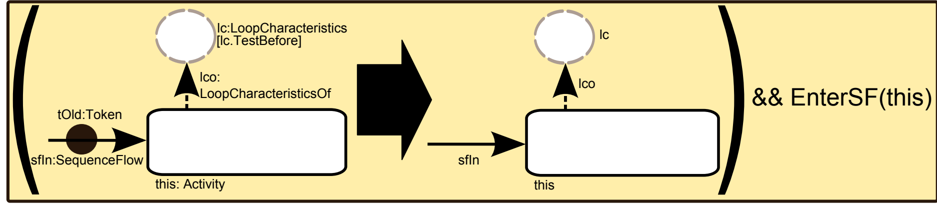


Figure 21: Corrected version of rule *skipLoopActivity*, fixing the error from Figure 20.

The token is transferred from an incoming sequence flow to an outgoing sequence flow directly. Although correct by intuition, this initial version overlooks the fact that specific activites (i.e., those of type *Task* can implicitly realize the *AND split* pattern. Figure 21 presents the rule that corrects this mistake: after removing the token from the incoming sequence flow, the rule delegates to rule *EnterSF*. This rule reuses helper rules that also support previously mentioned rules to realize the token transfer to the proper sequence flow(s).

More specifically, *EnterSF* includes rules *EnterSFone* and *EnterSFand-split*, which are used also by rules *leaveTaskMoreOut* and *leaveTaskOneOut*, as discussed in the context of Figures 12 and 13. Rule *EnterSF* only matches if either rule *EnterSFone* or rule *EnterSFandsplit* matches. By binding the match of a subrule to the identifier *e*, rule *EnterSF* can trigger the side-effects of its subrules in its own right-hand side. This is realized by the right-hand side statement *e()*.

As another example, consider helper rule *EnterSFone* on Figure 23. The

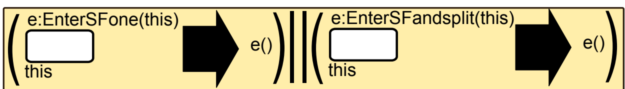


Figure 22: Helper rule *EnterSF(this:SequenceFlow)*.

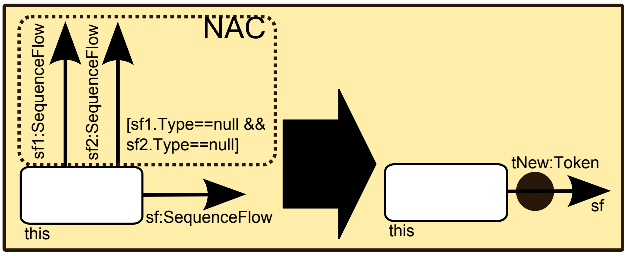


Figure 23: Helper rule *EnterSFone(this:Activity)*.

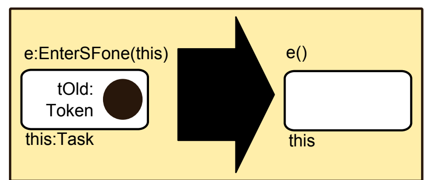


Figure 24: Rule *leaveTaskOneOut*, refactored version (reuse of *EnterSFone(this:Activity)*).

figure shows that rule *EnterSFone* very strongly resembles rule *leaveTaskOneOut*. Therefore, the latter rule actually also reuses *EnterSFone*. Figure 24 shows a refactored version of the rule: the refactored version reuses rule *EnterSFone* and as a result, it only needs to take care of the removal of the token from the input activity. The reuse is (again) realized by including in the left-hand side the expression *e:EnterSFone(this)*. This includes the left-hand side of rule *EnterSFone* into *leaveTaskOneOut*, and it binds the match of the subrule to *e*. The right-hand side contains the expression *e()*. This expression calls the right-hand side of the matched subrule while applying other side-effects (i.e, while removing *tOld*). Similar reuse techniques have been applied for eliminating code duplication between rule *leaveTaskMoreOut* and rules *skipLoopActivity*.Without elaborating further to a detailed discussion of these reuse mechanisms here, this illustrates that graph rewriting languages also enable the elimination of duplicated code.

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 Figures 25 and 26. Figure 25 shows that a parallel gateway can receive a token in case it has no incoming control flows that do not have a token. This double negation is equivalent to stating that all incoming control flows must have a

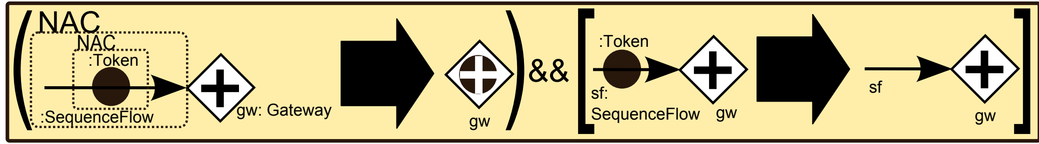


Figure 25: Enter a Parallel Gateway: rule *enterParallel*

token. If this condition is met, (1) the gateway receives a token and (2) a token is removed from each incoming control flow. The second part is realized by calling a subrule with maximal matching (i.e., with square brackets, "[ ]", around the call). Remark that it would be incorrect to embed the call in an *iterated* block, since then *all* tokens would be removed from the incoming edges (whereas just one token per incoming flow should be removed).

Remark that even with the maximal matching operator (i.e., with square brackets), each arc/token combination will still be matched just once, since matching is isomorphic by default. The rule as shown on Figure 25 executes BPMN processes correctly, even if they are not one-bounded (safe [2]). Although one could argue that in a workflow modeling context, business analysts *should* model only safe nets, they *may* model non-safe nets unconsciously, especially if they have little training in BPMN semantics. Therefore, it is important that our formalization (and related execution engine) can deal with non-safe nets too.

Figure 26 shows that if a parallel gateway has a token, it can put a token on each of its outgoing control flows. This rule again consists of two parts, one that indicates that the token on the gateway must be removed and one that indicates that a token must be put on each outgoing control flow. Also note that the rule is highly similar to rule *leaveTaskMoreOut* (cfr., Figure 12). This similarity is leveraged to reuse rule fragments at the implementation level (using the mechanisms described in the context of rules *leaveTaskMoreOut* and rule *skipLoopActivity* from Figures 12 and 21). We refrain from showing this reuse here, since it would pollute our documentation oriented rules with technical details.

Two rules define the behavior of the exclusive gateway as shown in Figure 27 and 28. Figure 27 shows that an exclusive gateway can receive a token when one of its incoming control flows has a token. When the gateway receives a token, the token on the incoming control flow is removed.

Figure 28 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

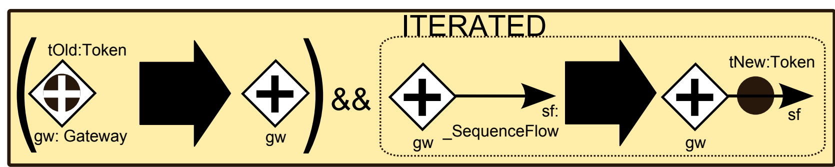


Figure 26: Leave a Parallel Gateway: rule *leaveParallelGateway*

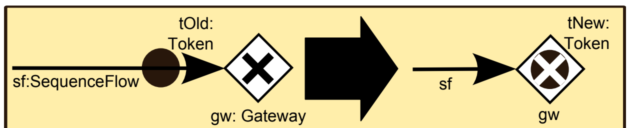


Figure 27: XOR-split activation: rule *enterExclusiveGateway*

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.

Rule *catchImplicitlyThrownException* (shown on Figure 29) formalizes a proposed extension to the BPMN 2.0 standard. Without this rule, exceptions that are thrown by 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., Figure 39).

Two rules define the behavior of the inclusive gateway as shown in Figure 30 and 33. Figure 30 shows when an inclusive gateway can receive a

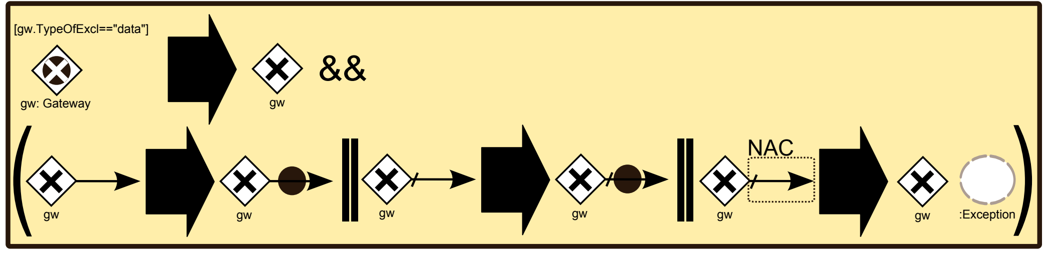


Figure 28: Transfer control from an XOR-split: rule *leaveExclusiveGateway*

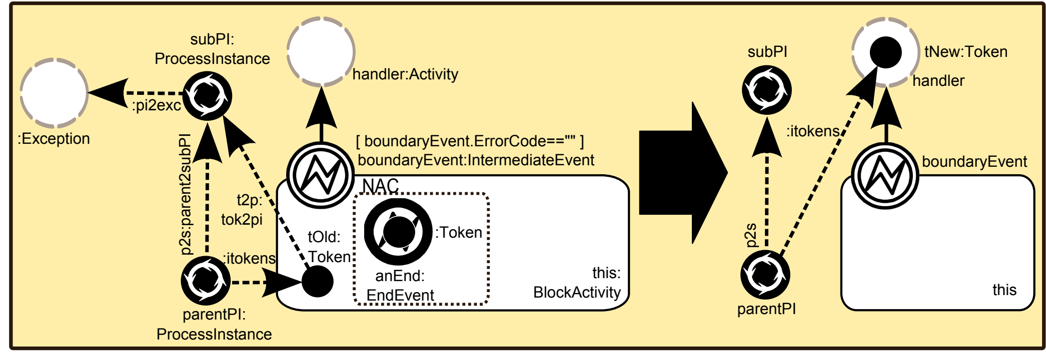


Figure 29: Handling exceptions that are thrown implicitly (i.e., *not* by a throw event)

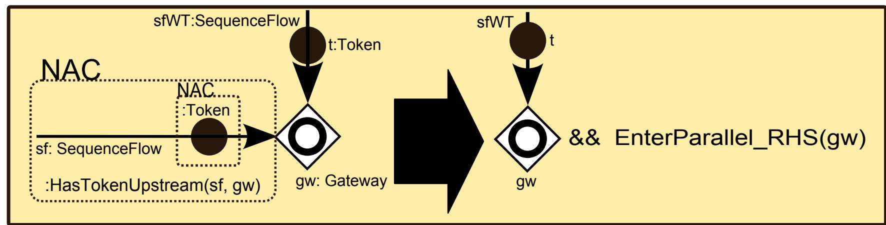


Figure 30: Enter an Inclusive Gateway: rule *enterInclusive*.

token. The right-hand side of rule *enterInclusive* delegates to the right-hand side of rule *enterParallel*. This ensures that a token is added to gateway *gw* and that a token is consumed from each incoming flow that has at least one token. 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 Figure 30) 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:

- 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 *HasViolating-TokenUpstream*. This helper pattern matches when there is a violing token

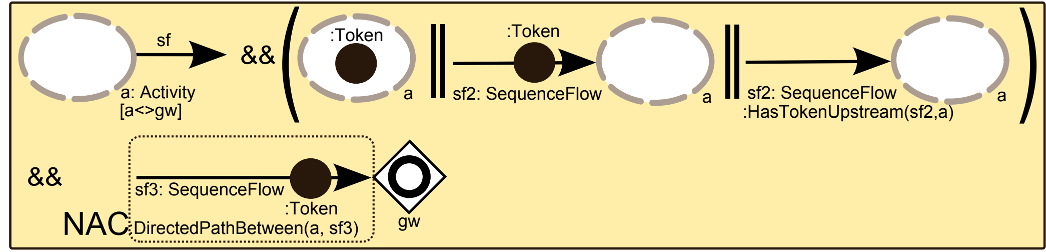


Figure 31: Helper pattern *HasViolatingTokenUpStream(sf:SequenceFlow, gw:Gateway)*.

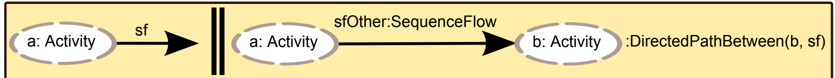


Figure 32: Helper pattern *DirectedPathBetween(a:Activity, sf:SequenceFlow)*.

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 *HasViolatingTokenUpstream* 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 Figure 31. The NAC applies a subpattern called *DirectedPathBetween* to match a transitive closure of the sequence flow edges between activity *a* and *sf3*, where *sf3* represents a non-empty incoming sequence flow of the gateway. Figure 32 show the definition of subpattern *DirectedPathBetween*. The transitive closure of sequence flow edges is realized by a recursive application of the subpattern.

Figure 33 shows what can happen when an inclusive gateway has a token: rule *leaveInclusive* is quite similar to rule *leaveExclusive* (cfr., Figure 28). It

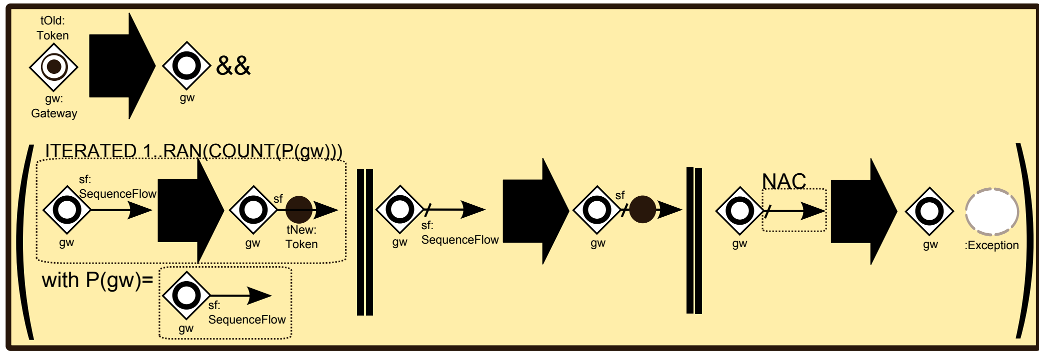


Figure 33: Leave an Inclusive Gateway: rule *leaveInclusive*.

also 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 that should be executed $1 \ldots RAN(n)$ times, which represents that the iterated block can be executed a number of times in 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). It should be noted that aggregation features ($COUNT$, AVG, etc.) are not yet widely supported by graph rewriting tools.

As indicated by Figure 8, 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 matches rule *leaveImplicit-InclusiveOut*. The latter rule is not shown here, but is straightforward, as it can reuse the right-hand side of rule *leaveInclusive*.

3.4. Events

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

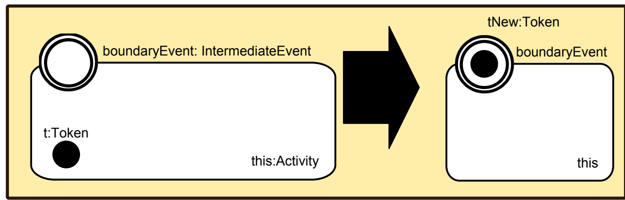


Figure 34: Enter an Autonomous Boundary Event: rule *enterAutonomousBoundaryEvent*.

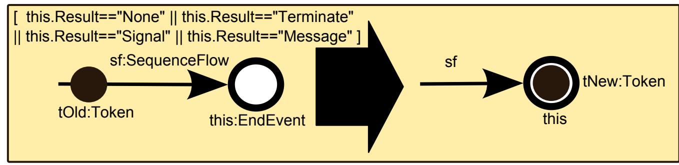


Figure 35: Entering various types of end events: rule *enterEndEvent*.

activity can continue. We focus on the first case (see Figure 34). The basic behavior of an end event is to simply receive a token (see Figure 35). Also note that rule *enterEndEvent* applies not only to regualr *end* events (i.e., those of result type *none*), but also to message events, etc.

Events can either catch a trigger (that may be thrown by others) or throw a result (that may be caught by others). BPMN 2.0 standardizes a number of triggers and results. The behavior of an event may differ, depending on the trigger that is caught or result that is thrown. We define the execution semantics for the message, error, compensation and signal events below. In addition to that triggers and results exist that do not receive special treatment in the control flow, but that may receive special treatment, because technical measures need to be taken to implement them. An example of this is the "timer" event that triggers when a preset moment in time is reached. In the control flow this does not lead to any special behavior, even though special measures must be taken in the implementation to catch the event at the specific moment at which the preset time is reached. Events that do not affect the control flow in a specific manner are timer, conditional, cancel, multiple and so-called "none" events, which, by definition, do not have a specific semantics.

Message Events. A message event can "throw" a message that can subsequently be "caught" by another message event. If a message throw event and a message catch event are connected by a message flow, the throw event

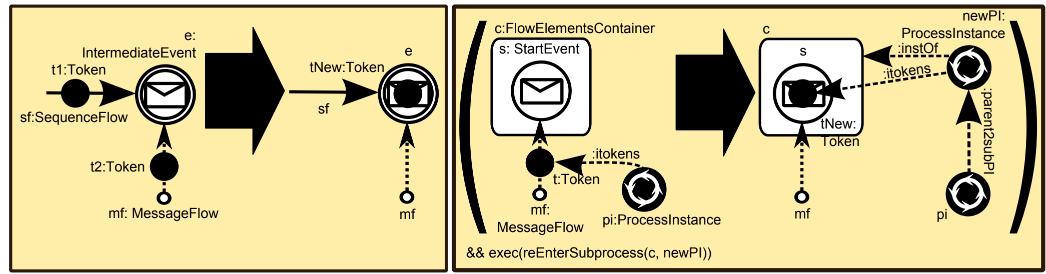


Figure 36: Rule *enterMessageCatchIntermediateEvent* and *enterMessageStartEvent*.

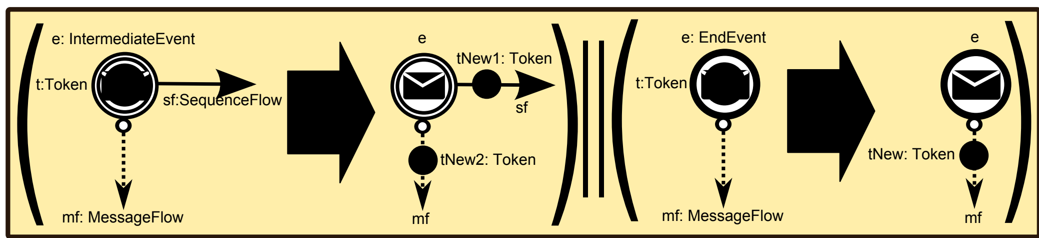


Figure 37: Throwing a message: rule *leaveMessageThrowEvent*.

must produce a message and the catch event must wait for that message to be produced. The message is passed along a message flow from the throw to the catch event. Figure 36 shows the rules for receiving messages: messages can be received either by message intermediate catch events or by message start events. The figure shows that such events can be entered, only if the message flow that points towards them also has a token.

Figure 37 shows the behavior of a message throw event, which can be either an intermediate or an end event. The figure shows that, upon leaving such events, a token is put on the outgoing message flow. In the case of an intermediate event, the outgoing sequence flow also receives a token.

As indicated by Figures 9 and 35, message throw events (as well as other kinds of throw events) are activated by rule *enterEndEvent*. Similarly, the rule *enterIntermediateThrowEvent* can activate message throw events as well as signal throw events. This rule is not shown as a figure in this paper, since it is so similar to rule *enterTask* (cfr., Figure 13).

Error Events. When an end event is reached by a token, the end event can "throw" an error. If it does that, the process instance that causes the error, reaches a state in which it is failed. Figure 38 shows rule *enterThrowErrorEvent*, which formalizes the activation of an end event with result type

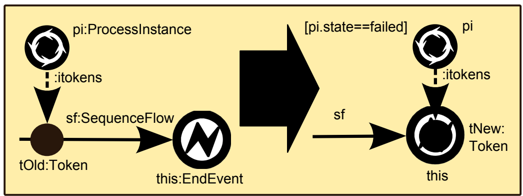


Figure 38: Throwing an error: rule *enterThrowErrorEvent*

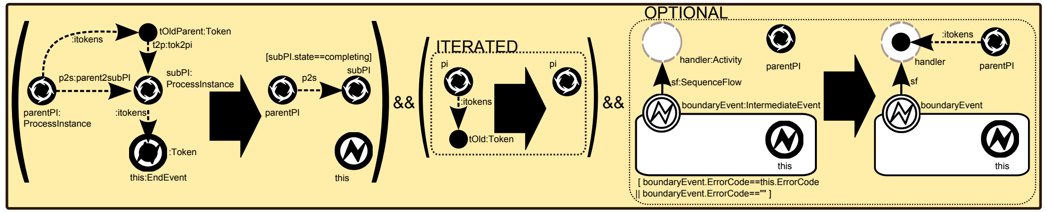


Figure 39: Handling errors: rule *leaveThrowErrorEvent*

error. Remark that the BPMN 2 standard does not support throwing errors from *intermediate* throw events, so we provide no rule for activating *ThrowIntermediateEvent* elements with an *error* trigger.

Figure 39 shows rule *leaveThrowErrorEvent*, which formalizes the error handling behavior. For one, the process instance that causes the error must be terminated. This is done by removing the token from the subprocess in which the error is thrown (cfr., *tOldParent* in Figure 39), removing the token from the error end event and removing all other tokens from the process instance (cfr., *tOld* in the *iterated* block in Figure 39). Although not shown on Figure 39, this can be realized by reusing the right-hand side of rule *completeProcess_RHS*, which has been discussed in the context of rule *completeProcessNormal* (cfr., Figure 10).

In case the subprocess in which the error occurs has a catch error event attached to it (cfr., the *optional* block in Figure 39), 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 Figure 29 for another case of *catch-all* behavior.

Compensation Events. Figures 40 and 41 show the rules for triggering a compensation. Both rules are the same, except for the type of the event node: besides the trivial move of the token from the input sequence flow to

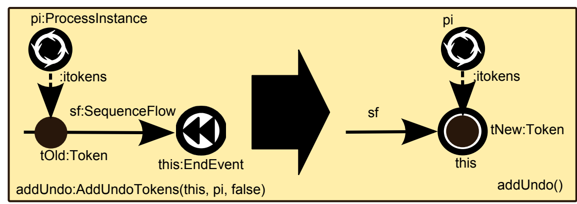


Figure 40: Initiating compensation (1/2): rule *enterCompensationEndEvent*.

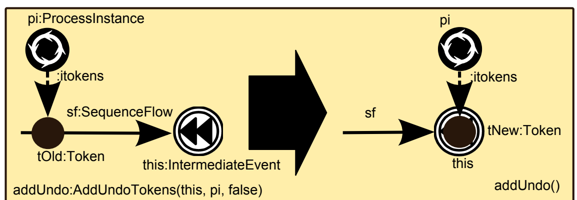


Figure 41: Compensation (2/2): rule *enterCompensationThrowIntermediateEvent*.

the event node, both rules include the subrule *AddUndoTokens*. That rule has two parameters: (1) the element *thrower* that throws the compensation event, and (2) the process instance *pi* that needs to be compensated.

AddUndoTokens recursively adds special tokens (elements of type *Undo-Token*) to the activities that need to be compensated. The rule also sets the status of the involved process instances to *compensating*. There are two cases for adding *UndoToken* elements in rule *AddUndoTokens*. Both cases are handled by subpatterns that are combined with the "‖|‖" operator.

The first subpattern deals with the compensation of one specific activity. Following that 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 that to be compensated activity gets an *UndoToken*.

The second subpattern 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. This behavior is realized by the nested *iterated* block in rule *AddUndoTokens*. The outer *iterated* block matches all so-called *history* tokens in the context of process instance *pi*. As indicated by the rewrite arrow in this block, each such match should produce an *undo token* in the same activity *actFromSameScope*. This realizes the compensation of all

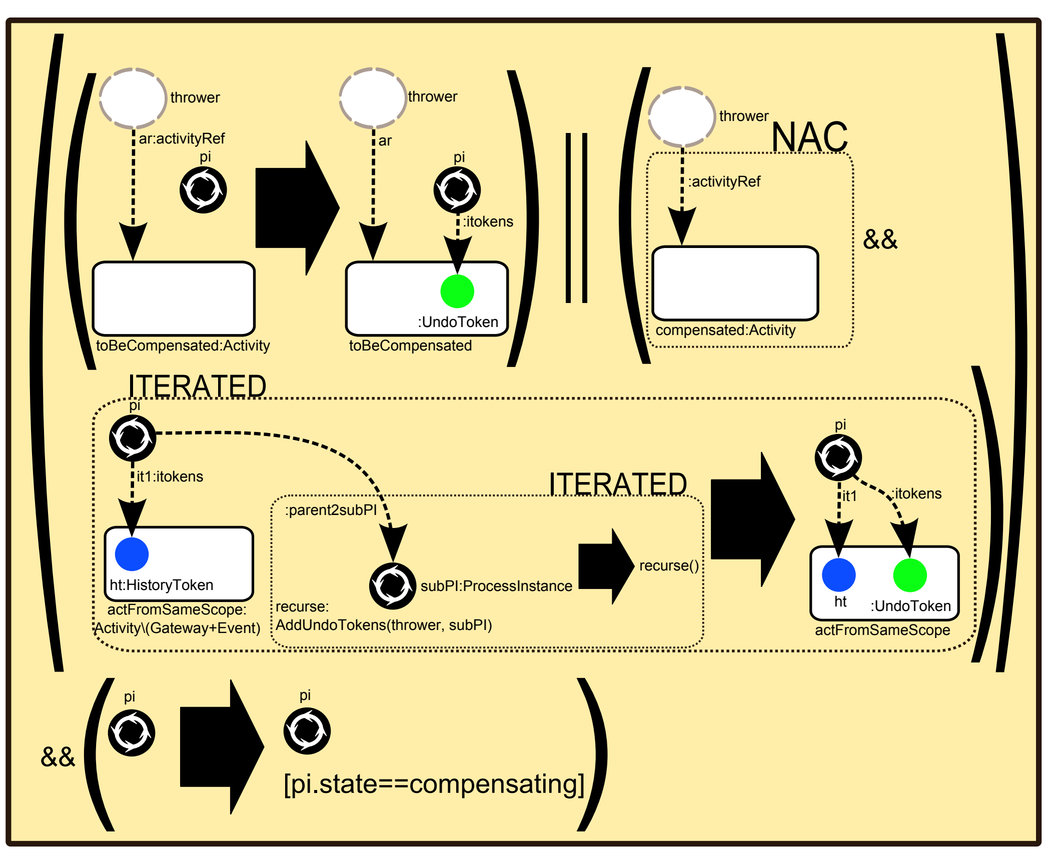


Figure 42: Compensating a subprocess recursively: helper rule *AddUndoTokens*.

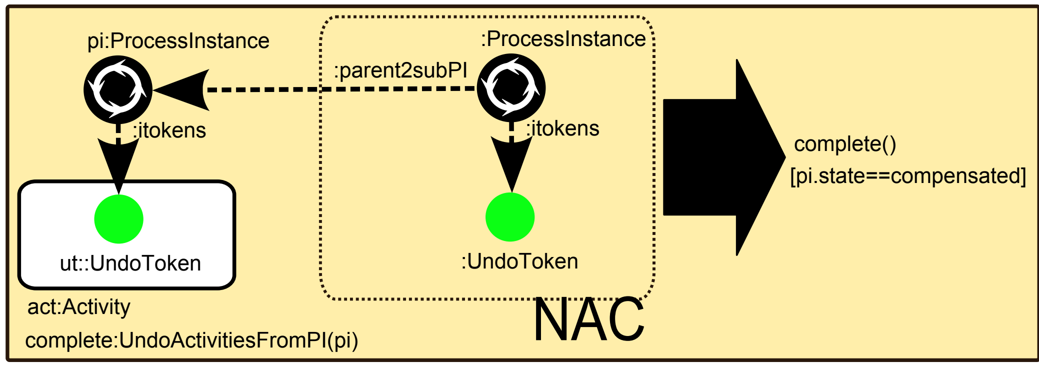


Figure 43: Compensation completion: rule *UndoProcessInstance*.

completed activities at one process instance level. Within this *iterated* block, there is a second *iterated* block. The latter block matches each subprocesses *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 they have been deliberately excluded from the metamodel in Section 2.3. 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.

Figures 43 and 44 show the rules for completing a compensation. Rule *UndoProcessInstance* matches a top-most process that contains an *UndoToken*. As explained in the context of Figures 40 and 41, such tokens represent ongoing compensation for activities that had completed. Rule *undoProcessInstance* sets the state of its matched process instance *pi* to *compensated* and includes helper rule *UndoActivitiesFromPI*, which 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 of *UndoActivitiesFromPI* ensures

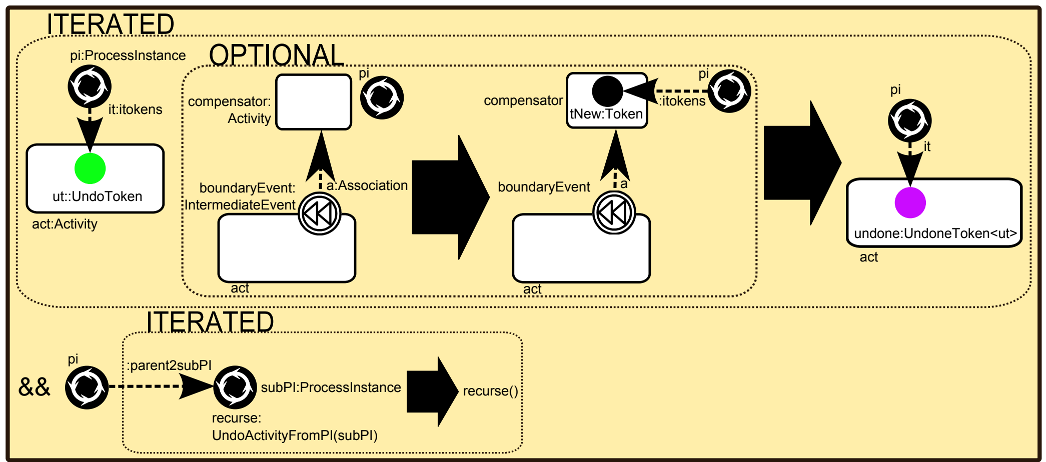


Figure 44: Compensation completion: rule *UndoActivitiesFromProcessInstance*.

that no parent process of *pi* could produce a match as well.

Helper rule *UndoActivitiesFromProcessInstance* models the compensation of each individual activity *act* in process instance *pi*. As indicated by rewrite variables *ut* and *undone*, this involves the re-typing of a node of type *UndoToken* to a node of type emphUndoneToken. 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*. Besides compensating each *act* in *pi*, rule *UndoActivitiesFromProcessInstance* matches all subprocesses of *pi* and evaluates there recursively. Remark that once a process is in the *compensated* state, the compensation is complete. In that case, the activity that triggered the compensation can be de-activated by rules *leaveCompensationThrowIntermediateEvent* or *completeProcessNormal*.

Signal Events. Signal events can be thrown by signal intermediate events or by signal end events. Subsequently, they can 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. Figure 45 and 46 show these two possible situations.

Rule *enterSignalCatchIntermediateEvent* (cfr., Figure 45) shows that an intermediate signal catch event can be entered in case there is a corresponding

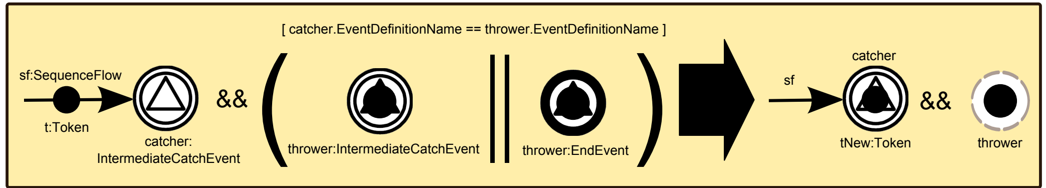


Figure 45: Reacting to a signal: rule *enterSignalCatchIntermediateEvent*.

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*.

Figure 46 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 scenario's: 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 rewrite 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 presents an implementation of the graph rewrite rules from section 3. It presents an implementation of the rules in a tool called Gr-

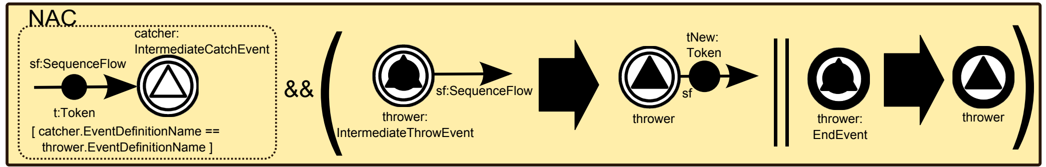


Figure 46: Dropping a signal: rule *leaveSignalThrowEvent*.

Gen.NET, an evaluation of the strengths and weaknesses of this implementation and possible alternatives to this implementation. In addition, it shows the traceability of the informal BPMN 2.0 execution semantics rules to GrGen.NET rules and it presents a use case of the GrGen.NET implementation as a reference implementation of the BPMN 2.0 execution semantics.

Subsections 4.1, 4.2 and 4.3 present the implementation, evaluation of the implementation and evaluation of alternatived, respectively. Subsection 4.4 shows the traceability and Subsection 4.5 shows how the implementation could be used as a reference implementation.

4.1. Implementation in GrGen.NET

An implementation of the rules from section 3 is made in a tool called GrGen.NET. The implementation is accessible from a web-based front-end as well as through various local GrGen.NET scripts[2]. It supports the following user scenario's:

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.

Batch Statespace Generation In this scenario, the rewrite 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 rewrite 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.

[2]See `http://is.tm.tue.nl/staff/rdijkman/bpmn.html`.

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 of the three scenarios. They are made available through an on-line virtual machine that contains (1) the GrGen.NET implementation of the rules from Section 3, (2) execution scripts for each of the three user scenario's outlined above, (3) a large collection of test models, (3) the version of GrGen.NET that should be used with the implementation, (4) an XPDL based BPMN editor, and (5) a tool to translate XPDL to the GrGen.NET input format. Using this virtual machine, the interested reader can evaluate the implementation with any model from the large collection of test models and with any other BPMN model that is modeled in the XPDL based BPMN editor. The GrGen.NET scripts are based on the GrGen.NET debugger and therefore do not visualize BPMN models in the BPMN standard concrete syntax. However, due to various configuration mechanisms, the graphs in the debugger do resemble that concrete syntax to a large extent.

Figure 47 shows an application of scenario one (*Manual Execution*), for the execution 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 of 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 of a particular rule: pressing "n" continues with a random match. Afterwards, pressing "e" will enable the user to enter the 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 on Figure 47 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

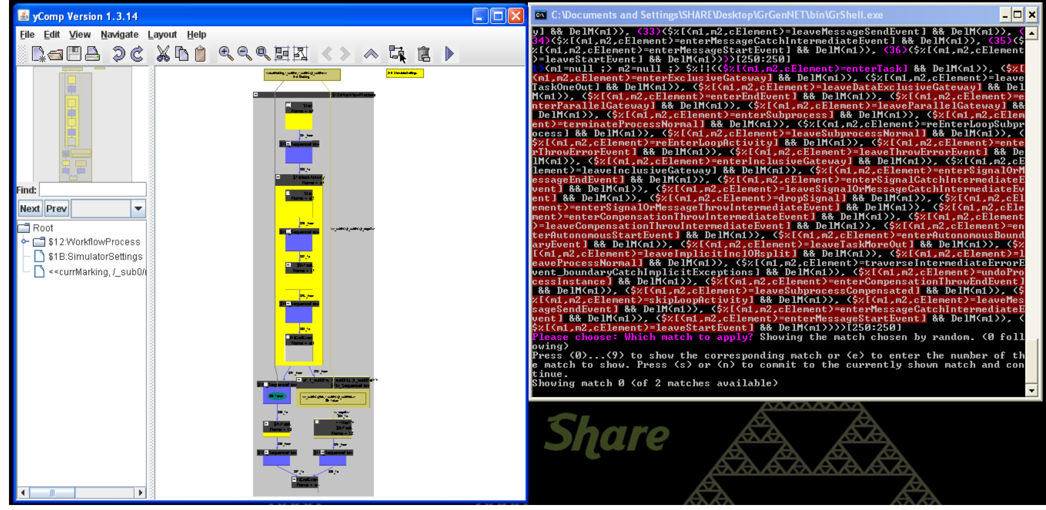


Figure 47: Screenshot of the *Manual Execution* mode.

rules (such as *enterExclusiveGateway* and *leaveDataExclusiveGateway*) have failed to match. The cyan text “enterTask” indicates that rule *enterTask* matches 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.

Figure 48 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 1GB of main memory and a mainstream CPU at the time of writing.

Figure 48 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

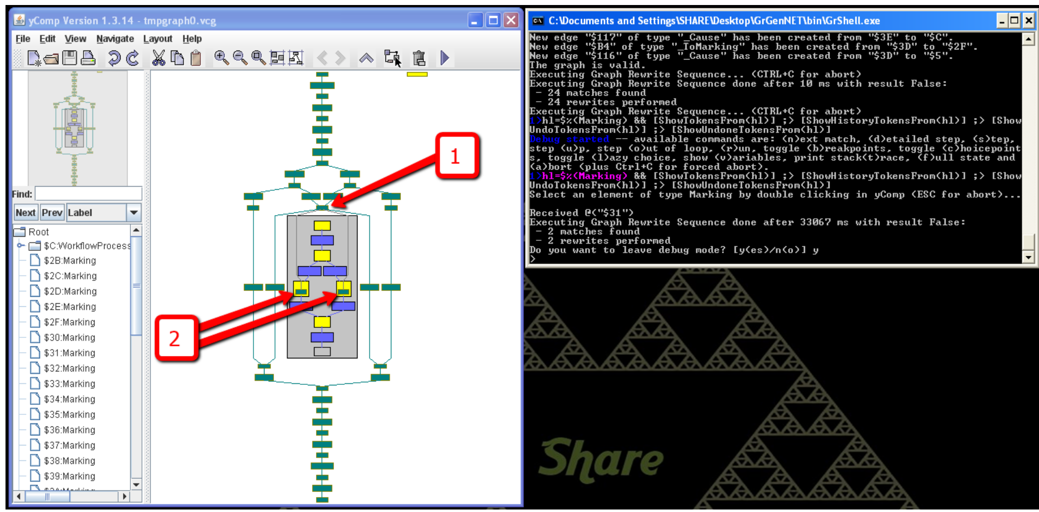


Figure 48: Screenshot of the *Batch Execution* mode and the (optional) marking visualizer.

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 here for one step and then merges in a common node. This is since the order of completion of tasks is arbitrary but after two steps, the second parallel gateway should hold a token.

The third execution mode (*Interactive Statespace Generation*) looks like a combination of Figures 48 and 47: after executing the operational semantics rules for a specific number of iterations, users run the rewriting system interactively and select a specific rewrite rule using the mechanisms shown on Figure 47.

4.2. Evaluation of the GrGen.NET implementation

Based on our experience with the implementation of the execution semantics in GrGen.NET, we identified the following strengths and weaknesses of GrGen.NET.
The GrGen.NET tool has the following strengths:

Visual Debugger GrGen.NET includes a visual debugging tool. We have used this tool to hunt down bugs by steping 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 change the level of detail. For example, in order to also visualize *ProcessInstance* nodes, just one configuration line needs to be adjusted. Within the graph rewriting domain, only Fujaba has comparably powerful debugging support. Remark that when no dedicated graph rewriting 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 [12, 4, 17]). 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 [34].

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 several bugs 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.

In contrast to these advantages, we have identified the following clear 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, Figure 49 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 rewrite rules for realizing scenario's two and three from Section 4.1 (i.e., scenario *Batch Statespace Generation* and *Interactive Statespace Generation*) contains rather complicated statespace construction logic. 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 rewriting tools such as Henshin [1] and GROOVE [25] 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 rewrite rules. Although this is adequate during programming, a visual representation seems more appropriate for documentation purposes. There are visual graph rewriting languages (see [27] for an overview) but most of these manipulate graphs in their abstract syntax representation. To the best of our knowledge, only the AToM^3 [18] approach enables the specification of rewrite rules in their concrete syntax form (i.e., in a form similar to the figures in section 3) but AToM^3 does not include the advanced iteration and negation operators that we have used throughout this work. Therefore, GrGen.NETwould provide unique functionality when providing an automatic translation of its textual rewrite rules to visual representations based on the concrete syntax of the domains under study (e.g., BPMN concrete syntax).

4.3. Alternatives to the GrGen.NET implementation

In summary, 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 are building a Henshin based prototype of a second implementation of the proposed rules. Besides Henshin's built-in support for statespace generation, the tool provides integration with Eclipse technologies such as EMF and GMF. These features should benefit the further dissemination of our reference semantics into industry.

It should be emphasized that the contribution of this paper is not specific to the domain of graph rewriting 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 rewriting languages and tools.

Then again, there are various graph rewriting tools that enable the embedding of graph rewriting programs within a general purpose programming language. A notorious example is the Fujaba tool, that supports the seemless integration of visual rewrite rules with Java statements. Various other tools (such as MoTMoT [19], AGG [3] and Henshin [1]) have adopted this approach and each of these tools has particular strengths and limitations. For the aforementioned reasons, we are giving Henshin an in-depth evaluation in our ongoing work.

4.4. Traceability, Mental Mapping to Standard

We claim that the formalization of the BPMN 2.0 execution semantics in terms of graph rewrite 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 rewrite rule can easily be found and the correctness of the graph rewrite rules can easily be checked. To an extent this traceability also applies to the implementation of the rewrite 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 [20]:

> *"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 Figure 12. 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: on the one hand, the text often omits information, since it is mentioned elsewhere already. On the other hand, the text sometimes repeats information that has been stated elsewhere already. In contrast, our implementation in the GrGen.NET language is designed to minimize code duplication. 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 traceabilitiy between our concise and executable implementation and the informal text from the standard.

Figure 49 shows a fragment of the GrGen.NET based implementation. The fragment contains the implementation of rule *leaveTaskMoreOut* (cfr., Figure 12) as well as various helper rules. Notice that the implementation of the rule takes just 14 (spaciously formatted) lines of code, the other code servers 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 Figure 12 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 (e.g., the use of *EnterSFandsplit* and *HasMultipleNullOutFlows*. Strong points of the implementation are that (1) it is also rule-based, (2) it leverages the same matching

```
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 }
```

Figure 49: Code fragment of the implementation, based on GrGen.NET syntax.

and control constructs as the visual rules and (3) all identifiers trace back to the conceptual discussion from 3. A point for improvement is that some details can be removed from the implementation, when using another graph rewriting tool. Therefore, the following subsection reflects on the suitability of GrGen.NET and describes our experience with alternatives.

4.5. Conformance Checking Architecture

For the execution semantics in terms of graph rewrite rules, we have shown a relatively good traceability to the execution semantics in the BPMN 2.0 specification and a relatively high level of completeness. In addition to that, we have shown above that the graph rewrite rules can be directly executed in a tool. These properties make the execution semantics in terms of graph rewrite rules ideal as a reference implementation of the execution semantics. The traceability makes the graph rewrite rules relatively easy to validate. The executability of the rules make them appropriate for comparing their behavior to the behavior of another tool that implements the execution semantics (e.g.: a workflow engine). The completeness makes it possible to do that for a large set of tools and features of those tools. Therefore, we propose to use the execution semantics by means of graph rewrite rules to verify implementations of the BPMN 2.0 execution semantics as follows.

Figure 50 shows that we use XPDL as the interchange format between BPMN modeling tools, workflow engines and the GrGen.NET implementation of the execution semantics. This is driven by the fact thats that (1) most BPMN tools are based on this format [8] and (2) that there are more than 50 XPDL implementations [21]. A BPMN model in XPDL can be imported both by various workflow engines and by our GrGen.NET implementation of the execution semantics. Consequently, the behavior of the workflow engines can then be compared to the behavior of the reference implementation to determine which workflow implementations (do not) implement the execution semantics correctly.

The comparison should be done by a conformance verification tool, of which the implementation is out of the scope of this paper. The verification tool does this by monitoring and controlling the behavior of both the workflow engine and the execution semantics and verifying that the workflow engine changes state in the same manner as the execution semantics. To this end the verification tool does the following:

1. Determine the execution traces that can be performed by the execution semantics.

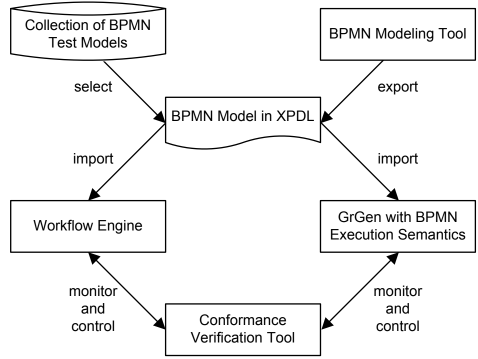


Figure 50: Conformance Testing of Workflow Engines

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

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

5. Related Work

The BPMN 2.0 standard specifies the complete BPMN 2.0 execution semantics in natural language. The use of natural language is sufficiently precise to allow for an intuitive understanding of the execution semantics, but it cannot be directly implemented into a tool for purposes of simulation, verification or execution. Therefore more precise semantics for BPMN have been defined [35, 36, 6, 23, 24, 7, 31]. 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 focus of the semantics and the BPMN constructs that are supported. Table 1 summarizes the means, goals and

Table 1: Means, Goals and Conceptual Focus of BPMN Semantics

Semantics	**Means**	**Goals**	**Focus**
BPMN 2.0 [20]	Natural language	Semantics specification	Complete
Wong and Gibbons [35, 36]	CSP	Refinement checking Property checking Soundness checking	Control-flow subset
Dijkman et al. [6]	Petri nets	Semantics specification Soundness checking	Control-flow subset
Prandi et al. [23]	COWS	Soundness checking Quantitative simulation	Control-flow subset Data-flow subset
Raedts et al. [24]	Petri nets	Soundness checking	Control-flow subset
Dumas et al. [7]	Pseudo code	Semantics specification	OR-Join
Takemura [31]	Petri nets	Semantics specification Soundness checking	Transactions
This paper	Graph rewriting	Semantics specification Conformance checking	Control-flow subset

conceptual focus of the semantics and Table 2 summarizes the features that are supported by the different semantics.

Wong and Gibbons [35, 36] define a semantics for a subset of the BPMN control-flow concepts in terms of the process algebra CSP [28]. 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 hours", and properties that apply to business process models in general, such as deadlock-freeness and proper completion [33]. We refer to the latter form of property checking as soundness checking. Dijkman et al. [6] define a semantics for a subset of the BPMN control-flow concepts in terms of classical Petri nets. The goal of their semantics is to define the BPMN execution semantics precisely and to enable soundness checking. Prandi et al. [23] define a semantics in terms of a process algebra called COWS [22]. Their semantics allows for soundness checking of BPMN models and also of quantitative simulation of BPMN models, provided that simulation information is provided with the BPMN model. The semantics is defined for a subset of both the control-flow and the data-flow aspect. Raedts et al. [24] 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. [7] define the execution semantics of a particular BPMN construct: the inclusive join gateway. Their goal is to discuss the execution semantics of this particularly complex construct in enough detail to allow animation of BPMN models that use this construct. Takemura [31] defines a semantics for the concepts that are related to BPMN transactions in terms of classical Petri nets. The goal of the semantics is to define the execution semantics of BPMN transactions precisely and to enable soundness checking.

The semantics in this paper differs from the other semantics with respect to the means that are used for the semantics, the completeness of the semantics and its prospective use. This paper uses graph rewrite rules to define the semantics. One benefit of using graph rewrite rules is that a direct mapping is possible from the execution semantics rules in the BPMN specification to graph rewrite rules. This direct mapping makes the graph rewrite rules easily traceable to BPMN execution semantics rules and easily understandable. Another benefit of using graph rewrite rules is their relative expressive power. For example, classical Petri nets are inherently limited in the semantics that they can represent; it is notoriously hard to represent the OR-join in classical Petri nets and data-related concepts cannot be represented in a feasible manner in classical Petri nets. Such concepts can easily be represented in graph rewriting systems. Table 2 supports this claim, by showing that the formal semantics that is presented in this paper is relatively complete and includes some notoriously hard concepts that are the sole focus of [7, 31]. These properties make our semantics particularly suited to be used for conformance verification. In particular it enables us to compare the execution of a running workflow system to the execution semantics as it is executed in a graph rewriting tool, where the execution semantics is relatively complete and its correctness is easily traceable.

6. Conclusion

This paper proposes a formalization of the BPMN 2.0 execution semantics in terms of graph rewrite rules. The paper shows that it is relatively easy to develop a complete formalization of the execution semantics in this way. It does that, both by showing that currently a large part of the BPMN 2.0 execution semantics rules are formalized (including some notoriously hard to formalize concepts such as the inclusive merge gateway and process compensation) and by showing that a larger part of the BPMN 2.0 execution

semantics rules is implemented than in the formalizations in related work. The paper also shows that such a formalization maintains good traceability to the informal execution semantics rules from which it is derived, because: (1) each rewrite rule can be traced back to a rule in the informal execution semantics and vice versa; (2) the structure of the graph rewrite rules follows the structure of the informal execution semantics; and (3) the graph rewrite rules can be represented graphically, using the BPMN 2.0 notation.

The formalization in terms of graph rewrite rules can be implemented. In this paper we show an implementation in the graph rewriting tool GrGen.NET. We show that this implementation can be used for various purposes, including simulation of a BPMN 2.0 model and generation of the state space of such a model with the purpose of doing (partial) state space analysis. 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 rewrite rules relatively easy to validate. The executability of the rules make them appropriate for comparing their behavior to the behavior of other tools (e.g., workflow engines). The completeness makes it possible to do that for a large set of tools and language features.

Based on our experience with implementing the graph rewrite rules in GrGen.NET, we conclude that the benefits of this tool are that it: is fast, has a visual debugger and is actively maintained. The drawbacks are that it neither supports the graphical style of specifying the graph transformation rules that facilitates traceability, nor supports direct state-space generation, such that specific graph rewrite rules need to be included to generate that state-space. Consequently, alternatives to an implementation in GrGen.NET can be explored. Currently, we are looking into an implementation in a competing tool called Henshin, but an implementation directly in a programming language, such as Java, can also be envisioned.

Concluding, we claim that the formalization of an execution semantics in general, and BPMN 2.0 in particular, in terms of graph rewrite rules is a powerful tool for various purposes. In particular it is suited for model simulation, model state space exploration and as a reference implementation. However, to fully exploit the benefits, more research can be done into features that are required of graph rewriting tools, when using them for formalization of execution semantics. In particular state space generation features and graphical rule representations – *in the concrete syntax of the target language (BPMN or others)* – should be considered.

References

[1] Thorsten Arendt, Enrico Biermann, Stefan Jurack, Christian Krause, and Gabriele Taentzer. Henshin: Advanced concepts and tools for in-place emf model transformations. In Dorina C. Petriu, Nicolas Rouquette, and Øystein Haugen, editors, *MoDELS*, volume 6394 of *Lecture Notes in Computer Science*, pages 121–135. Springer, 2010.

[2] E. Best and P. S. Thiagarajan. *Some classes of live and safe Petri nets*, pages 71–94. Springer-Verlag New York, Inc., New York, NY, USA, 1987.

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

[4] Sebastian Buchwald and Moritz Kroll. A GrGen.NET solution of the antworld case for the grabats 2008 contest. In A. Rensink and P Van Gorp, editors, *4th International Workshop on Graph-Based Tools: The Contest*, 2008.

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

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

[7] Marlon Dumas, Alexander Grosskopf, Thomas Hettel, and Moe Wynn. Semantics of standard process models with or-joins. In *Proceedings of OTM 2007, Part I*, volume 4803 of *Lecture Notes in Computer Science*, pages 41–58, 2007.

[8] Philip Effinger, Martin Siebenhaller, and Michael Kaufmann. An interactive layout tool for bpmn. *E-Commerce Technology, IEEE International Conference on*, 0:399–406, 2009.

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

[10] Hartmut Ehrig, Reiko Heckel, Martin Korff, Michael Löwe, Leila Ribeiro, Annika Wagner, and Andrea Corradini. Algebraic approaches to graph transformation - part ii: Single pushout approach and comparison with double pushout approach. In *Handbook of Graph Grammars*, pages 247–312, 1997.

[11] Hartmut Ehrig, Michael Pfender, and Hans Jürgen Schneider. Graph-grammars: An algebraic approach. In *FOCS*, pages 167–180. IEEE, 1973.

[12] Rubino Geißand Moritz Kroll. On improvements of the varro benchmark for graph transformation tools. Technical Report 2007-7, Universität Karlsruhe, IPD Goos, 12 2007. ISSN 1432-7864.

[13] A. Habel and D. Plump. Computational completeness of programming languages based on graph transformation. In *Proc. FoSSaCS 2001*, volume 2030 of *Lecture Notes in Computer Science*, page 230, 2001.

[14] Annegret Habel, Reiko Heckel, and Gabriele Taentzer. Graph grammars with negative application conditions. *Fundamenta Informaticae*, 26:287–313, June 1996.

[15] Annegret Habel and Karl-heinz Pennemann. Correctness of high-level transformation systems relative to nested conditions. *Mathematical. Structures in Comp. Sci.*, 19(2):245–296, 2009.

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

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

[18] Juan de Lara and 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, pages 174–188, London, UK, UK, 2002. Springer-Verlag.

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

[20] Object Management Group. Business process model and notation (BPMN) – version 2.0 (formal/2011-01-03). `http://www.omg.org/spec/BPMN/2.0/PDF`, 2011.

[21] Ludmila Penicina. Towards the mapping of multidimensional bpmn models to process definition standards. *Journal of Riga Technical University*, 41:76–83, 2010.

[22] D. Prandi and P. Quaglia. Stochastic COWS. In *Proceedings of ICSOC 2007*, volume 4749 of *Lecture Notes in Computer Science*, pages 245–256, 2007.

[23] Davide Prandi, Paola Quaglia, and Nicola Zannone. Formal analysis of BPMN via a translation into COWS. In *Proceedings of COORDINATION 2008*, volume 5052 of *Lecture Notes in Computer Science*, pages 249–263, 2008.

[24] I.G.J. Raedts, M. Petkovic, Y.S. Usenko, J.M.E.M. van der Werf, J.F. Groote, and 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*, pages 126–137. INSTICC Press, 2007.

[25] Arend Rensink. The GROOVE simulator: A tool for state space generation. In *Applications of Graph Transformations with Industrial Relevance (AGTIVE)*, volume 3062 of *Lecture Notes in Computer Science*, pages 479–485. Springer, 2003.

[26] Arend Rensink. Nested quantification in graph transformation rules. In Andrea Corradini, Hartmut Ehrig, Ugo Montanari, Leila Ribeiro, and Grzegorz Rozenberg, editors, *Graph Transformations*, volume 4178

of *Lecture Notes in Computer Science*, pages 1–13. Springer Berlin / Heidelberg, 2006.

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

[28] A. W. Roscoe. *The Theory and Practice of Concurrency*. Prentice-Hall, 1998.

[29] Andy Schürr. Programmed graph transformations and graph transformation units in grace. In Janice Cuny, Hartmut Ehrig, Gregor Engels, and Grzegorz Rozenberg, editors, *Graph Grammars and Their Application to Computer Science*, volume 1073 of *Lecture Notes in Computer Science*, pages 122–136. Springer Berlin / Heidelberg, 1996.

[30] Bruce Silver. *BPMN Method and Style: A levels-based methodology for BPM process modeling and improvement using BPMN 2.0*. Cody-Cassidy Press, June 2009.

[31] Tsukasa Takemura. Formal semantics and verification of BPMN transaction and compensation. In *Proceedings of the IEEE Asia-Pacific Conference on Services Computing*, pages 284–290, Los Alamitos, CA, USA, 2008. IEEE Computer Society.

[32] Wil M. P. van der Aalst. Workflow patterns. In Ling Liu and M. Tamer Özsu, editors, *Encyclopedia of Database Systems*, pages 3557–3558. Springer US, 2009.

[33] W.M.P. van der Aalst. Verification of workflow nets. In *Proceedings of the 18th International Conference on Application and Theory of Petri Nets*, pages 407–426, 1997.

[34] Pieter Van Gorp and Rik Eshuis. Transforming process models: Executable rewrite rules versus a formalized java program. In Dorina Petriu, Nicolas Rouquette, and ystein Haugen, editors, *Model Driven Engineering Languages and Systems*, volume 6395 of *Lecture Notes in Computer Science*, pages 258–272. Springer Berlin / Heidelberg, 2010.

[35] Peter Y. Wong and Jeremy Gibbons. A process semantics for BPMN. In *Proceedings of the 10th International Conference on Formal Methods*

and Software Engineering, volume 5256 of *Lecture Notes In Computer Science*, pages 355–374, 2008.

[36] Peter Y.H. Wong and Jeremy Gibbons. Formalisations and applications of BPMN. *Science of Computer Programming*, In Press, Corrected Proof, 2009.

Appendix A

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,
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 $\mathcal{A}_{ev} \subseteq \mathcal{A}$,

     - $\mathcal{E}_{name}$ 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 signallers 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 block 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[3],
  - $\mathcal{T}_{before}$: $\mathcal{L}_{std} \rightarrow \mathcal{D}_{bool}$ 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,

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

- $\rightarrow_{BPMN}$: $\mathcal{M} \times \mathcal{M}$ 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*}.

Table 2: Features Supported by BPMN Semantics

Feature	BPMN Standard [20]	Wong & Gibbons [35, 36]	Dijkman et al. [6]	Prandi et al. [23]	Raedts et al. [24]	Dumas et al. [7]	Takemura [31]	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							

Working Papers Beta 2009 - 2011

nr.	Year	Title	Author(s)
353	2011	BOMN 2.0 Execution Semantics Formalized as Graph Rewrite Rules: extended version	Pieter van Gorp, Remco Dijkman
352	2011	Resource pooling and cost allocation among independent service providers	Frank Karsten, Marco Slikker, Geert-Jan van Houtum
351	2011	A Framework for Business Innovation Directions	E. Lüftenegger, S. Angelov, P. Grefen
350	2011	The Road to a Business Process Architecture: An Overview of Approaches and their Use	Remco Dijkman, Irene Vanderfeesten, Hajo A. Reijers
349	2011	Effect of carbon emission regulations on transport mode selection under stochastic demand	K.M.R. Hoen, T. Tan, J.C. Fransoo G.J. van Houtum
348	2011	An improved MIP-based combinatorial approach for a multi-skill workforce scheduling problem	Murat Firat, Cor Hurkens
347	2011	An approximate approach for the joint problem of level of repair analysis and spare parts stocking	R.J.I. Basten, M.C. van der Heijden, J.M.J. Schutten
346	2011	Joint optimization of level of repair analysis and spare parts stocks	R.J.I. Basten, M.C. van der Heijden, J.M.J. Schutten
345	2011	Inventory control with manufacturing lead time flexibility	Ton G. de Kok
344	2011	Analysis of resource pooling games via a new extention of the Erlang loss function	Frank Karsten, Marco Slikker, Geert-Jan van Houtum
343	2011	Vehicle refueling with limited resources	Murat Firat, C.A.J. Hurkens, Gerhard J. Woeginger
342	2011	Optimal Inventory Policies with Non-stationary Supply Disruptions and Advance Supply Information	Bilge Atasoy, Refik Güllü, TarkanTan
341	2011	Redundancy Optimization for Critical Components in High-Availability Capital Goods	Kurtulus Baris Öner, Alan Scheller-Wolf Geert-Jan van Houtum
339	2010	Analysis of a two-echelon inventory system with two supply modes	Joachim Arts, Gudrun Kiesmüller

338	2010	Analysis of the dial-a-ride problem of Hunsaker and Savelsbergh	Murat Firat, Gerhard J. Woeginger
335	2010	Attaining stability in multi-skill workforce scheduling	Murat Firat, Cor Hurkens
334	2010	Flexible Heuristics Miner (FHM)	A.J.M.M. Weijters, J.T.S. Ribeiro
333	2010	An exact approach for relating recovering surgical patient workload to the master surgical schedule	P.T. Vanberkel, R.J. Boucherie, E.W. Hans, J.L. Hurink, W.A.M. van Lent, W.H. van Harten
332	2010	Efficiency evaluation for pooling resources in health care	Peter T. Vanberkel, Richard J. Boucherie, Erwin W. Hans, Johann L. Hurink, Nelly Litvak
331	2010	The Effect of Workload Constraints in Mathematical Programming Models for Production Planning	M.M. Jansen, A.G. de Kok, I.J.B.F. Adan
330	2010	Using pipeline information in a multi-echelon spare parts inventory system	Christian Howard, Ingrid Reijnen, Johan Marklund, Tarkan Tan
329	2010	Reducing costs of repairable spare parts supply systems via dynamic scheduling	H.G.H. Tiemessen, G.J. van Houtum
328	2010	Identification of Employment Concentration and Specialization Areas: Theory and Application	F.P. van den Heuvel, P.W. de Langen, K.H. van Donselaar, J.C. Fransoo
327	2010	A combinatorial approach to multi-skill workforce scheduling	Murat Firat, Cor Hurkens
326	2010	Stability in multi-skill workforce scheduling	Murat Firat, Cor Hurkens, Alexandre Laugier
325	2010	Maintenance spare parts planning and control: A framework for control and agenda for future research	M.A. Driessen, J.J. Arts, G.J. v. Houtum, W.D. Rustenburg, B. Huisman
324	2010	Near-optimal heuristics to set base stock levels in a two-echelon distribution network	R.J.I. Basten, G.J. van Houtum
323	2010	Inventory reduction in spare part networks by selective throughput time reduction	M.C. van der Heijden, E.M. Alvarez, J.M.J. Schutten
		The selective use of emergency shipments for	

322	2010	service-contract differentiation	E.M. Alvarez, M.C. van der Heijden, W.H. Zijm
321	2010	Heuristics for Multi-Item Two-Echelon Spare Parts Inventory Control Problem with Batch Ordering in the Central Warehouse	
320	2010	Preventing or escaping the suppression mechanism: intervention conditions	B. Walrave, K. v. Oorschot, A.G.L. Romme
319	2010	Hospital admission planning to optimize major resources utilization under uncertainty	Nico Dellaert, Jully Jeunet.
318	2010	Minimal Protocol Adaptors for Interacting Services	R. Seguel, R. Eshuis, P. Grefen.
317	2010	Teaching Retail Operations in Business and Engineering Schools	Tom Van Woensel, Marshall L. Fisher, Jan C. Fransoo.
316	2010	Design for Availability: Creating Value for Manufacturers and Customers	Lydie P.M. Smets, Geert-Jan van Houtum, Fred Langerak.
315	2010	Transforming Process Models: executable rewrite rules versus a formalized Java program	Pieter van Gorp, Rik Eshuis.
314	2010	Getting trapped in the suppression of exploration: A simulation model	Bob Walrave, Kim E. van Oorschot, A. Georges L. Romme
313	2010	A Dynamic Programming Approach to Multi-Objective Time-Dependent Capacitated Single Vehicle Routing Problems with Time Windows	S. Dabia, T. van Woensel, A.G. de Kok
	2010		
312	2010	Tales of a So(u)rcerer: Optimal Sourcing Decisions Under Alternative Capacitated Suppliers and General Cost Structures	Osman Alp, Tarkan Tan
311	2010	In-store replenishment procedures for perishable inventory in a retail environment with handling costs and storage constraints	R.A.C.M. Broekmeulen, C.H.M. Bakx
310	2010	The state of the art of innovation-driven business models in the financial services industry	E. Lüftenegger, S. Angelov, E. van der Linden, P. Grefen
309	2010	Design of Complex Architectures Using a Three Dimension Approach: the CrossWork Case	R. Seguel, P. Grefen, R. Eshuis
308	2010	Effect of carbon emission regulations on transport mode selection in supply chains	K.M.R. Hoen, T. Tan, J.C. Fransoo, G.J. van Houtum
307	2010	Interaction between intelligent agent strategies for real-time transportation planning	Martijn Mes, Matthieu van der Heijden, Peter Schuur

306	2010	Internal Slackening Scoring Methods	Marco Slikker, Peter Borm, René van den Brink
305	2010	Vehicle Routing with Traffic Congestion and Drivers' Driving and Working Rules	A.L. Kok, E.W. Hans, J.M.J. Schutten, W.H.M. Zijm
304	2010	Practical extensions to the level of repair analysis	R.J.I. Basten, M.C. van der Heijden, J.M.J. Schutten
303	2010	Ocean Container Transport: An Underestimated and Critical Link in Global Supply Chain Performance	Jan C. Fransoo, Chung-Yee Lee
302	2010	Capacity reservation and utilization for a manufacturer with uncertain capacity and demand	Y. Boulaksil; J.C. Fransoo; T. Tan
300	2009	Spare parts inventory pooling games	F.J.P. Karsten; M. Slikker; G.J. van Houtum
299	2009	Capacity flexibility allocation in an outsourced supply chain with reservation	Y. Boulaksil, M. Grunow, J.C. Fransoo
298	2010	An optimal approach for the joint problem of level of repair analysis and spare parts stocking	R.J.I. Basten, M.C. van der Heijden, J.M.J. Schutten
297	2009	Responding to the Lehman Wave: Sales Forecasting and Supply Management during the Credit Crisis	Robert Peels, Maximiliano Udenio, Jan C. Fransoo, Marcel Wolfs, Tom Hendrikx
296	2009	An exact approach for relating recovering surgical patient workload to the master surgical schedule	Peter T. Vanberkel, Richard J. Boucherie, Erwin W. Hans, Johann L. Hurink, Wineke A.M. van Lent, Wim H. van Harten
295	2009	An iterative method for the simultaneous optimization of repair decisions and spare parts stocks	R.J.I. Basten, M.C. van der Heijden, J.M.J. Schutten
294	2009	Fujaba hits the Wall(-e)	Pieter van Gorp, Ruben Jubeh, Bernhard Grusie, Anne Keller
293	2009	Implementation of a Healthcare Process in Four Different Workflow Systems	R.S. Mans, W.M.P. van der Aalst, N.C. Russell, P.J.M. Bakker
292	2009	Business Process Model Repositories - Framework and Survey	Zhiqiang Yan, Remco Dijkman, Paul Grefen
291	2009	Efficient Optimization of the Dual-Index Policy Using Markov Chains	Joachim Arts, Marcel van Vuuren, Gudrun Kiesmuller
290	2009	Hierarchical Knowledge-Gradient for Sequential Sampling	Martijn R.K. Mes; Warren B. Powell; Peter I. Frazier
289	2009	Analyzing combined vehicle routing and break scheduling from a distributed decision making perspective	C.M. Meyer; A.L. Kok; H. Kopfer; J.M.J. Schutten
288	2009	Anticipation of lead time performance in Supply Chain Operations Planning	Michiel Jansen; Ton G. de Kok; Jan C. Fransoo
287	2009	Inventory Models with Lateral Transshipments: A Review	Colin Paterson; Gudrun Kiesmuller; Ruud Teunter; Kevin Glazebrook
286	2009	Efficiency evaluation for pooling resources in health care	P.T. Vanberkel; R.J. Boucherie; E.W. Hans; J.L. Hurink; N. Litvak

285	2009	A Survey of Health Care Models that Encompass Multiple Departments	P.T. Vanberkel; R.J. Boucherie; E.W. Hans; J.L. Hurink; N. Litvak
284	2009	Supporting Process Control in Business Collaborations	S. Angelov; K. Vidyasankar; J. Vonk; P. Grefen
283	2009	Inventory Control with Partial Batch Ordering	O. Alp; W.T. Huh; T. Tan
282	2009	Translating Safe Petri Nets to Statecharts in a Structure-Preserving Way	R. Eshuis
281	2009	The link between product data model and process model	J.J.C.L. Vogelaar; H.A. Reijers
280	2009	Inventory planning for spare parts networks with delivery time requirements	I.C. Reijnen; T. Tan; G.J. van Houtum
279	2009	Co-Evolution of Demand and Supply under Competition	B. Vermeulen; A.G. de Kok
278	2010	Toward Meso-level Product-Market Network Indices for Strategic Product Selection and (Re)Design Guidelines over the Product Life-Cycle	B. Vermeulen, A.G. de Kok
277	2009	An Efficient Method to Construct Minimal Protocol Adaptors	R. Seguel, R. Eshuis, P. Grefen
276	2009	Coordinating Supply Chains: a Bilevel Programming Approach	Ton G. de Kok, Gabriella Muratore
275	2009	Inventory redistribution for fashion products under demand parameter update	G.P. Kiesmuller, S. Minner
274	2009	Comparing Markov chains: Combining aggregation and precedence relations applied to sets of states	A. Busic, I.M.H. Vliegen, A. Scheller-Wolf
273	2009	Separate tools or tool kits: an exploratory study of engineers' preferences	I.M.H. Vliegen, P.A.M. Kleingeld, G.J. van Houtum
272	2009	An Exact Solution Procedure for Multi-Item Two-Echelon Spare Parts Inventory Control Problem with Batch Ordering	Engin Topan, Z. Pelin Bayindir, Tarkan Tan
271	2009	Distributed Decision Making in Combined Vehicle Routing and Break Scheduling	C.M. Meyer, H. Kopfer, A.L. Kok, M. Schutten
270	2009	Dynamic Programming Algorithm for the Vehicle Routing Problem with Time Windows and EC Social Legislation	A.L. Kok, C.M. Meyer, H. Kopfer, J.M.J. Schutten
269	2009	Similarity of Business Process Models: Metics and Evaluation	Remco Dijkman, Marlon Dumas, Boudewijn van Dongen, Reina Kaarik, Jan Mendling
267	2009	Vehicle routing under time-dependent travel times: the impact of congestion avoidance	A.L. Kok, E.W. Hans, J.M.J. Schutten
266	2009	Restricted dynamic programming: a flexible framework for solving realistic VRPs	J. Gromicho; J.J. van Hoorn; A.L. Kok; J.M.J. Schutten;

Working Papers published before 2009 see: http://beta.ieis.tue.nl