

Verification of Business Process Quality Constraints Based on Visual Process Patterns

Alexander Förster, Gregor Engels, Tim Schattkowsky
Dept. of Computer Science
University of Paderborn
Warburger Strasse 100, 33098 Paderborn, Germany
Email: {alfo,engels,timschat}@upb.de

Ragnhild Van Der Straeten
System and Software Engineering Lab
Vrije Universiteit Brussel
Pleinlaan 2, 1050 Brussel, Belgium
Email: rvdstrae@vub.ac.be

Abstract

Business processes usually have to consider certain constraints like domain specific and quality requirements. The automated formal verification of these constraints is desirable, but requires the user to provide an unambiguous formal specification. In particular since the notations for business process modeling are usually visual flow-oriented languages, the notational gap to the languages usually employed for the formal specification of constraints, e.g., temporal logic, is significant and hard to bridge. Thus, our approach relies on UML Activities as a single language for the specification of both business processes and the corresponding constraints. For the expression of such constraints, we have provided a process pattern definition language based on specialized Activities. In this paper, we describe how model checking can be employed for formal verification of business processes against such patterns. For this, we present an automated transformation of the business process and the corresponding patterns into a transition system and temporal logic, respectively.

1 Introduction

Effective and reliable business processes are a major building block for the success of modern enterprises. However, such business processes and their corresponding models can become very complex. For the design, understanding, and maintenance of big and complex processes it is necessary that *process constraints*, i.e., properties and requirements related to the processes, can be verified.

In the context of business processes, such requirements are for example legal, domain specific, or quality requirements. In particular with the rising popularity of modern Total Quality Management (TQM) systems, the question whether such quality requirements are fulfilled by a busi-

ness process becomes increasingly important.

The quality requirements contained in a TQM system are usually given in natural language and are therefore difficult to be checked against existing business processes. This is also the usual case for other domain specific requirements in organizations. In order to make requirements for processes manageable and enable automated verification, they need to be specified in a precise, formal way such that exact methods can be applied to verify the correct fulfillment of the specified requirements by given business processes. Furthermore, it should also be possible to specify quality requirements such that they are easy to formulate, read, and to apply by humans like quality managers, domain experts, and process designers.

These two different demands are at first sight contradictory. One way out of this dilemma is to define a language that allows specifying quality requirements in a user-friendly way, yet having a clear formal underpinning. In previous works, we have already proposed an approach for modeling process constraints, in particular related to quality management, that is based on *process patterns* [7]. These process patterns can be visually modeled using a subset of UML2.0 Activities [14] with light-weight extensions based on stereotypes called the *Process Pattern Specification Language (PPSL)*. A complementary pattern-based development process and some further extensions of the PPSL can be found in [8].

UML Activities have become a widespread modeling language for business processes. Such Activities are usually represented as Activity Diagrams. Many process developers are familiar with the syntax of Activity Diagrams and their meaning. The PPSL as an extension to UML Activities allows process developers to also model process constraints based on the language they are familiar with.

In previous works we have described the construction of business processes with respect to process patterns reflecting quality constraints. Based on that, we propose the ver-

ification of existing processes to ensure conformance to a given set of process constraints. The constraints are visually modeled as a process pattern using the PPSL. However, the definition of a precise meaning of the PPSL elements is a necessary prerequisite to allow verification of the process constraints. Therefore, we will define the semantics of the PPSL by presenting a translation of PPSL models into temporal logic.

The next section discusses related work before section 3 introduces the PPSL together with a small example. In Sect. 4, we present our approach in three consecutive steps. First, we formalize the behavior of the business process as a labeled transition system (LTS). Second, we provide an explicit translation of the PPSL elements into temporal logic. Finally, we show how the temporal logic formulas can be checked against the LTS representation by a state-of-the-art model checker. We describe a preliminary tool chain implemented as an integrated workbench to facilitate the design and verification process for the business process designer. Thus, our approach supports the business process designer in determining if the behavior of the business process conforms to the requirements specified by a process pattern.

2 Related Work

For the topic of modeling constraints for business processes using comprehensible visual notation consistent with that of the business process, the related work falls into these categories: workflow and process patterns, checking formal properties of workflows and processes, modeling behavioral predicates, and Activity Diagram semantics.

Van der Aalst *et al.* [18] have devised a number of workflow patterns concerning different types of control flows in workflow systems. Their aim was mainly to demonstrate the expressiveness and capabilities of existing workflow management systems and workflow specification languages. Unlike our approach, their process patterns cover mainly technical concepts like all kinds of different basic and complex control flows and they are focused on Petri nets. Approaches like [1] and [16] consider the application of process patterns to software development processes. However, these approaches cover aspects specific to software development and contain no formal underpinning or means for automatic checking.

There are approaches checking formal properties of workflow models like Van der Aalst and Kindler [13]. These approaches focus on formally verifying general properties like soundness, fairness, termination etc. In contrast to that our approach allows the verification of user-defined, specific properties like quality management requirements or domain specific requirements.

In [3], Deveraux and Chechik present an approach for building behavioral models of event-driven systems. These

models can then be verified over a given software program to conform to certain kinds of temporal or causal properties. However, the approach assumes that the software program is already presented as a Kripke structure and thus remains at an abstract level. For our application area, this is not sufficient as both the transformation of the actual application language into such structures is not elaborated and the definition of the properties to be checked is too general. Thus, our approach employs similar basic ideas, but at a different granularity and up to the level of the real application language including a generic approach on defining process constraints.

In [12], the authors propose an approach for model checking of business processes using temporal logic. The approach is based on a proprietary process modeling language. The authors provide formalizations for some basic sorts of constraints. However, there is no support for user-defined constraints.

To allow model checking of UML Activity Diagrams, we need to employ a formal semantics for Activities. In [17], the authors provide a translation into Petri Nets. However, this translation does not consider some important semantic properties of Activities like *traverse-to-completion* etc. Also the non-local semantics of some model elements like *ActivityFinalNodes* is not covered in this approach. In [5], the author presents an in-depth coverage of a translation of UML Activities into the input language of NuSMV. This semantics description is based on UML 1.x finite state machine semantics for Activities. UML 2 Activities have a completely different semantics based on token flow. Unfortunately, the translation is therefore not applicable.

In [6], we have introduced the general idea of using patterns to describe quality requirements for business processes. In [7], we introduced a pattern language and built the ground for a formulation of an abstract pattern-instance relationship for process patterns. In this paper, we focused on specifying the formal semantics of process patterns for automatic conformance checking. Related work includes general works on the application of process patterns as well as the verification of behavioral properties in such processes.

3 Quality Assurance in Business Processes

In this section we use as an example a business process that is a slightly adopted version of one of the example processes in the UML Specification [14, p. 312], as shown in Fig. 1. To briefly recapitulate the PPSL, we state some domain specific and quality management requirements and model them using the PPSL. In succeeding sections we will show how the corresponding process patterns can be translated into temporal logic and verified against given Activity Diagram based processes. As a first process constraint we

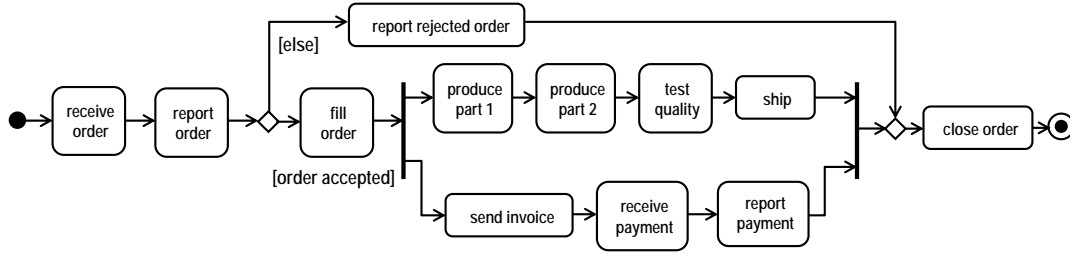


Figure 1. Example business process (adopted from [14, p. 312])

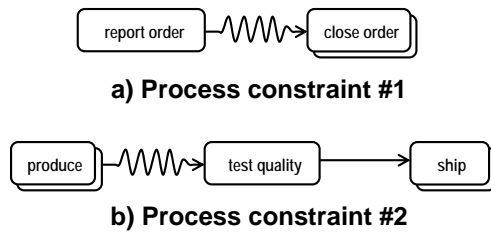


Figure 2. Process patterns for constraints #1 and #2

can state:

Process constraint #1: Before an order is being closed, records of the received orders have to be made.

The constraint implies that the Action “report order” is executed at some point before the Action “close order” is executed, but it does not require that the Action “report order” is executed *directly* before “close order”.

It is an important property of typical process requirements that they frequently contain rather loose or incomplete temporal/logical relationships between Actions. In a concrete business process there may be many other Actions executed in between “report order” and “close order” without contradicting the pattern. Since the original semantics of an ActivityEdge as described in the UML Superstructure is that Action “close order” is enabled immediately when Action “report order” terminates [14], we introduced the stereotype $\ll\text{after}\gg$ for an ActivityEdge to express that some Action has to be executed after another but not necessarily directly following it. Stereotyping of model elements is the standard extension mechanism of the UML. Using stereotypes, model elements can be given additional or extended semantics. Figure 2a shows process constraint #1 modeled in our PPSL. The curly line in Fig. 2a is a visualization option of the $\ll\text{after}\gg$ stereotype. In the remainder, we refer to this sort of stereotyped ActivityEdge as *AfterEdge*.

Being able to express such loose order relationships in process patterns is also a necessary prerequisite to enable

flexible application of the process patterns since pattern actions and actions of the original business process usually need to be weaved together. If the pattern designer wants to specify that there may not be other Actions being executed in between two Actions of a pattern, a regular ActivityEdge without stereotype can be used in the pattern.

Process constraint #1 could be read in two directions. Either “every time an order is closed this has to be preceded by reporting an order” or “every report of an order must be followed by closing the order”. It is important to have the possibility to distinguish these two cases in the process constraint language. This can be done using the stereotype $\ll\text{all}\gg$ for Actions. It denotes whether the implication given by the AfterEdge in the constraint refers to *all* “close order” Actions or *all* “report order” Actions. In the remainder, we will refer to an Action having an $\ll\text{all}\gg$ stereotype as *AllAction*. The multi-node in Figs. 2 and 3 are a visualization option of the AllAction. It is also possible to use AllActions on both sides of the AfterEdge or ActivityEdge denoting that both implications have to be fulfilled. Consequently, it is a well-formedness rule for our language that at least one of two Actions being connected by an AfterEdge or ActivityEdge is an AllAction.

The next process constraint that we want to consider is:

Process constraint #2: After each production action a quality check has to be performed prior to delivery.

Process constraints #2 is similar to process constraint #1 but contains precisely spoken two different constraints put together. The first requirement is that after each production action there has to be a quality check and the second requirement is that before shipping a product, the quality has to be checked. This is why the actions “produce” and “ship” in the process pattern are AllNodes. The use of a regular ActivityEdge between “test quality” and “ship” sets the requirement that shipping has to be *directly* preceded by the quality test. There may not be other actions executed in between these two actions.

If we now compare the process constraints with the example business process in Fig. 1, we can see that it does not have an action called “produce” like the pattern in Fig.

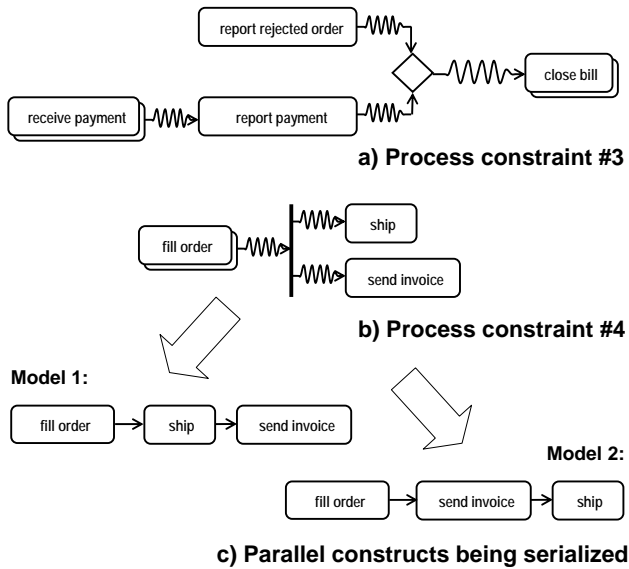


Figure 3. Process pattern for constraints #3 and #4

2b. In particular if the pattern and the business process have been developed by different persons, it will frequently be the case that actions having different names refer to the same behavior. Therefore, a mapping between the action names of the business process and the process pattern based on the action’s behaviors has to be established prior to the verification of the process constraints. In our case, “produce part 1” and “produce part 2” of the business process are both mapped to “produce” of the pattern.

Now we want to consider the following process constraints to demonstrate some additional aspects of the PPSL:

Process constraint #3: Before an order is being closed, either records of payments made or records of the fact that the order was rejected have to be taken. Each payment received shall be reported.

Process constraint #4: When an order is filled, a product has to be shipped and an invoice has to be sent.

Figure 3a shows process constraint #3. It demands that one of the two Actions “report rejected order” or “report payment” has to be performed before the bill is being closed while “report payment” has to be executed after a payment was received. Conditional control flows, modeled by Fork-, Join, Decision- and MergeNodes, can be used in the PPSL like in regular Activity Diagrams to express such constraints.

Process constraint #4 is shown in Fig. 3b. Parallel control flows in the pattern mean that the actions of these control flows may be executed concurrently. When the pattern is applied, the parallel control flows should generally

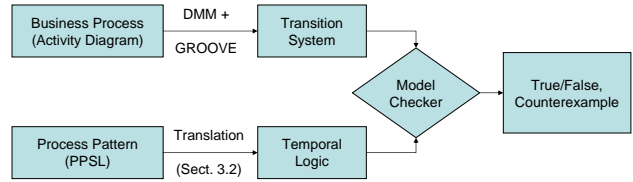


Figure 4. Verification process.

be sustained in the resulting process. However, parallelism in the process patterns can just be an expression of the fact that the order in which Actions are executed is irrelevant. Accordingly, in a business process any valid interleaving between the concurrent actions of the pattern is a correct application of the pattern as shown in Fig. 3c. This conforms to the semantics of parallel control flows described in the UML 2.0 Superstructure that real parallelism is not enforced.

In the remainder we want to give a precise, formal notion to when a business process model conforms to a process pattern and therefore respects the process constraints, such as quality requirements, encoded in the pattern.

4 Formalization

The principal aim is to be able to check whether a concrete business process conforms to a given process pattern. Fig. 4 describes the employed verification process.

To perform the conformance check we first need to specify the exact execution semantics of the given business process. Then, we need to precisely define how patterns modeled in the PPSL constrain this business process to finally be able to check these process patterns.

To provide the execution semantics of the business processes, we use the Dynamic Meta-Modeling (DMM) framework developed at the University of Paderborn [4]. The DMM framework is a semantics description method for visual modeling languages in general which combines a denotational meta modeling framework for expressing static semantics with operational rules capturing the behavior of the elements. For a detailed description of the DMM approach we refer to [4]. DMM has been successfully applied to statecharts [4], to UML 1.x sequence diagrams [11], and to UML 2.0 Activity Diagrams [10]. We briefly explain this approach in the next section and we show how the resulting interpretation of business process models is utilized in our approach.

As shown in the previous section, the process patterns specify logical and temporal constraints over the business process. Therefore, the semantics of the PPSL is provided by temporal logic formulas. We provide an explicit translation from a process pattern into temporal logic formulas. This translation is defined and exemplified in Sect. 4.2.

Please note that the PPSL is designed to allow modeling, formalizing and verifying constraints for business processes, it is not intended to be a graphical notation for temporal logic in general.

We show how the concrete example process patterns of Sect. 3 are translated into temporal logic and whether the business process of Fig. 1 conforms to these patterns. Finally in Sect. 4.3 we show how the verification process shown in Fig. 4 can be embedded in a tool chain, using state-of-the-art model checkers. This tool support supports the business process designer in verifying the application of the process patterns he/she selected.

4.1 Generation of the Labeled Transition System

The semantics of a visual language is defined in the DMM framework by a semantic domain meta model and a set of meta operations. The semantic domain meta model describes the semantical concepts of the language. For example, to be able to express the semantics of Activity Diagrams, the semantic concept `ActionExecution` is defined as a class in the semantic domain meta model. This concept denotes a currently running execution of an Action. For each semantic concept that relates to behavior it captures this behavior in a set of meta operations. The meta operations are defined by rules represented as UML communication diagrams. These communication diagrams are given a formal interpretation based on graph transformation rules.

Given the set of DMM rules for a particular language and a user-defined model expressed in the same language, a labeled transition system (LTS) is generated by a DMM interpreter that reflects all possible behaviors to the model. In the DMM approach, the GROOVE (GRaphical Object-Oriented VERification) tool set [15] has been chosen as DMM interpreter to produce the resulting labeled system.

Using the GROOVE tool, the set of DMM rules for UML 2.0 Activity Diagrams, and given a user-defined Activity Diagram, which is in our case the business process, we can generate a LTS that specifies the exact execution paths of the Activity Diagram. Figure 5 shows an excerpt of the resulting LTS from the example in Fig. 1. Each state in the LTS represents a state in the execution of the Activity Diagram. The labels in the states represent the fact that the corresponding Action is actually executing.

A name of an Action in the business process refers to a certain *Behavior*. Since the business process and the process pattern may have been devised by different persons using different Behavior namespaces, a mapping needs to be defined. This mapping is part of the tool chain described in Sect. 4.3. For the formalization, without loss of generality, we assume that the Behavior namespaces are synchronized.

The DMM approach for UML Activity Diagrams incor-

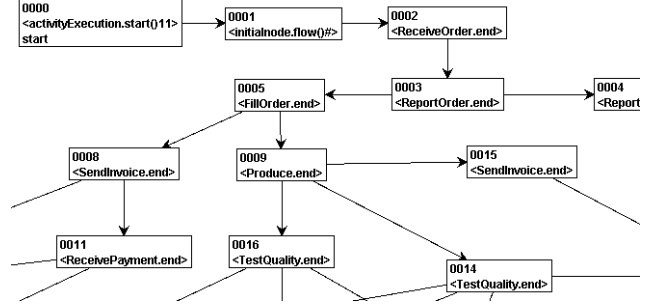


Figure 5. Excerpt of the transition system resulting from the example business process in Fig. 1b generated using DMM and GROOVE.

porates the semantics of the UML 2.0 Superstructure [14] for model elements of the packages *StructuredActivities* and *IntermediateActivities*. These semantics implemented in DMM include all important issues described in the UML Specification like traverse-to-completion, the fact that Actions capture all of their input tokens in one atomic step, etc. Concurrency in the Activity Diagram leads to a transition system that contains all possible interleavings between the concurrent Actions.

In the next section we specify how the process patterns can be translated into temporal logic formulas which can be checked against the transition system.

4.2 Formalization of Process Patterns

The formalization of the process patterns is presented in two consecutive steps. First, the notion of *Pattern Graph* is defined. Secondly, the translation of a process pattern into temporal logic is established.

A process pattern is represented by a *Pattern Graph*.

Definition 1. A **Pattern Graph (PG)** is a tuple $PG = (N, E)$ where N is the set of nodes and E is the set of edges, i.e., the set of tuples $N \times N$. The notation $e(n_1, n_2)$ is equivalent with $e \in E \wedge e = (n_1, n_2)$.

The set N is divided into different disjoint subsets, $N = N_a \cup N_d \cup N_m \cup N_f \cup N_j$ where N_a is the set of ActionNodes, N_d is the set of DecisionNodes, N_m is the set of MergeNodes, N_f is the set of ForkNodes, and N_j is the set of JoinNodes.

The set of ControlNodes, denoted by N_c , is defined as $N_c = N_d \cup N_m \cup N_f \cup N_j$.

The set of edges E is divided into two disjoint sets $E = E_d \cup E_a$, where

E_d is the set of ActivityEdges, E_a is the set of AfterEdges, and $E_d \cap E_a = \emptyset$.

The set of AllActions is denoted by $N_{all} \subseteq N_a$.

In the remainder of this section, we explain how process

patterns can be expressed by *Linear-time temporal logic* (LTL) formulas. LTL has appropriate expressional power for the formalization of the semantics of the PPSL. We use the temporal connectives F to denote “some Future state”, G to denote “all future states (Globally)”, X to denote “the neXt state”. We use LTL with the past operators O to denote “previously” and Y to denote “the previous state”. The use of LTL with past operators makes the formulation of some of the formulas significantly shorter and more intuitive. It shall be noted that past operators do not increase the complexity of LTL model checking and can be equivalently converted to future-only LTL [9].

The translation of the pattern into temporal logic formulas is defined recursively. We first determine the translation of the basic PPSL elements into temporal logic formulas as shown in Tab. 1. Using this recursive translation, the translation of a pattern graph corresponding to a process pattern is defined.

Actions. Let $a \in \mathbf{N}_a$, the Action is translated to a proposition. This proposition corresponds to the name representing the Action in the transition system representing the business process under study. The fixed set of propositions considered for the translation of the patterns is the set of the action names occurring in the generated transition system as described in Section 4.1.

Two Actions are connected through an Edge. We specify as a well-formedness rule of the pattern graph that there has to be an AllAction at one side at least of an Edge, i.e.,

$$\forall e = (n_1, n_2) \in \mathbf{E} : n_1 \in \mathbf{N}_{all} \vee n_2 \in \mathbf{N}_{all} \quad (1)$$

Row 2 to 4 in Tab. 1 each show the three possible ways how Actions can be connected by an AfterEdge (conforming to the well-formedness rule (1)) and their respective translation to a LTL formula.

The LTL formula $G(a \rightarrow F b)$ (row 2, column 3) expresses that each time a is executed it is eventually followed by the execution of b . The formula $G(b \rightarrow O a)$ expresses that if an execution of b exists, it has to be preceded by the execution of a . The conjunction of both LTL formulas states the meaning of an AllAction Node a connected to an AllAction Node b through an AfterEdge.

As an example consider process constraint #1 (cf. Sect. 3). This requirement will be translated to the following LTL formula:

$$G(close_order \rightarrow O report_order) \quad (2)$$

The AfterEdge specified in this pattern spans nearly the whole business process. For the business process of Fig. 1 to fulfill this constraint it is important that the alternative and parallel parts of the business process are all merged and joined properly. Thus it is guaranteed that the execution of the business process finally reaches “close order” at

some point after “report order”, so formula 2 holds. Checking process constraint #1 has some interesting implications. Say the business process designer wants to make an alteration to the business process such that if the quality check fails, the process should be terminated. Figure 6 shows the alteration in the process model. The semantics of the FinalNode as described in the UML Specification is that all tokens in the Activity that is executed will be terminated immediately. The transition system resulting from the DMM transformation reflects this behavior. Accordingly, formula 2 will evaluate to false after the alteration meaning that it is now not guaranteed anymore that the order will be closed. If somebody had put the alteration shown in Fig. 6 somewhere in the middle of a much bigger business process, such deficiencies would probably be much harder to detect manually.

Quality Requirement #2 results in two LTL formulas which both have to be fulfilled.

$$G(produce \rightarrow F test_quality) \quad (3)$$

$$G(ship \rightarrow Y test_quality) \quad (4)$$

Similar to the case where two Actions are connected through an AfterEdge, three cases can be distinguished where two Actions are connected through a regular ActivityEdge. Again an AllAction can be followed by an AllAction or an AllAction can be followed by an Action or an Action can be followed by an AllAction through an Edge. How the three different constraints are translated is shown in row 5 to 7 in Tab. 1. These LTL formulas are equal to the LTL formulas representing the corresponding kinds of Actions connected through an AfterEdge, except that the temporal connectives O and F are replaced Y and X , respectively.

The question now arises how control nodes in the pattern graph have to be interpreted. Table 2 provides for each control node a small example pattern fragment and the general translation to LTL formulas in case of AfterEdges connected to the control node. In the remainder of this section, we explain the translation of the control nodes in detail.

There are two additional well-formedness rules for the use of control nodes. For the DecisionNode and the ForkN-

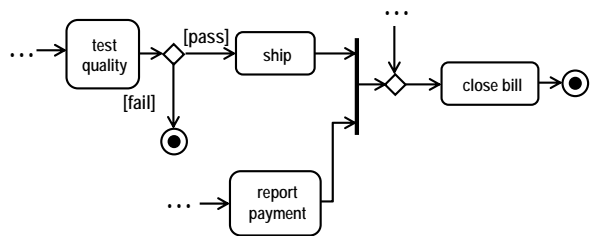
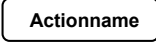



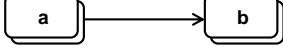
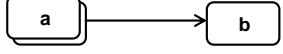
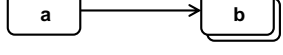


Figure 6. Alteration to the business process of Fig. 1

Table 1. Translation of the PPSL elements into LTL.

Model element	Notation	Translation
Action		ActionName
AfterEdge between AllActions		$G(a \rightarrow F b) \wedge G(b \rightarrow O a)$
AfterEdge between AllAction and Action		$G(a \rightarrow F b)$
AfterEdge between Action and AllAction		$G(b \rightarrow O a)$
ActivityEdge between AllActions		$G(a \rightarrow X b) \wedge G(b \rightarrow Y a)$
ActivityEdge between AllAction and Action		$G(a \rightarrow X b)$
ActivityEdge between Action and AllAction		$G(b \rightarrow Y a)$

ode, either the Action preceding the control node has to be an AllAction or all nodes following the control node have to be AllActions (5). For the MergeNode and the JoinNode, a similar well-formedness rule applies in the opposite direction (6).

$$e(a, c) \in \mathbf{E} \wedge c \in \mathbf{N}_d \cup \mathbf{N}_f \wedge \forall_{i=1, \dots, n} : e_i(c, b_i) \Rightarrow$$

$$a \in \mathbf{N}_{\text{all}} \vee \forall_{i=1, \dots, n} : b_i \in \mathbf{N}_{\text{all}} \quad (5)$$

$$\forall_{i=1, \dots, n} : e_i(a_i, c) \wedge c \in \mathbf{N}_m \cup \mathbf{N}_j \wedge e(c, b) \in \mathbf{E} \Rightarrow$$

$$\forall_{i=1, \dots, n} : a_i \in \mathbf{N}_{\text{all}} \vee b \in \mathbf{N}_{\text{all}} \quad (6)$$

As already explained in case of two Actions connected by an edge, generally different formulas have to be created depending on whether the node(s) preceding or succeeding the control node are AllActions. Therefore, for each control node we will explain two cases.

DecisionNodes Let us at first assume all edges are AfterEdges. If $a \in \mathbf{N}_{\text{all}}$ and $\forall i \in \{1, \dots, n\} : b_i \notin \mathbf{N}_{\text{all}}$, the corresponding pattern is translated to the LTL formula $G(a \rightarrow \bigvee_{i=1, \dots, n} F b_i)$. This formula expresses whenever an Action a is executed, eventually at least one of the b_i ($i \in \{1, \dots, n\}$) will be executed, reflecting the choice semantics of the DecisionNode. If some b_i nodes are also

AllActions, this means that the execution of these b_i Actions need to be eventually preceded by the execution of an a Action. This implies that a formula $G(b_i \rightarrow O a)$ needs to be added for each $b_i \in \mathbf{N}_{\text{all}}$. If a is not an AllAction but only the b_i nodes are AllActions (remark that in this case, following our well-formedness rule it is mandatory that all b_i nodes are AllActions) the corresponding pattern is translated to the LTL formula $\bigwedge_{i=1, \dots, n} G(b_i \rightarrow O a)$, only.

Regular ActivityEdges can also be used to connect an Action with a DecisionNode and vice versa. There are always two edges of the pattern involved in each subpart of the resulting formula, i.e., the edge from Action a to the DecisionNode and the Edge from the DecisionNode to b_i . If both edges are regular ActivityEdges the translations as specified in Tab. 2 have to be changed by replacing the temporal connective O by Y and F by X . If at least one of the two edges is an AfterEdge, it shall overrule the regular ActivityEdge and the temporal connectives F and O remain. This does not only hold for DecisionNodes but for each ControlNode.

MergeNodes Again with $m \in \mathbf{N}_m$ we make a distinction between the case where $\forall_{i=1, \dots, n} : e_i(a_i, m) \wedge a_i \in \mathbf{N}_{\text{all}}$ and the case where $e(m, b) \in \mathbf{E} \wedge b \in \mathbf{N}_{\text{all}}$. The first case

Table 2. Translation of the control nodes into LTL

Model element	Example	General Translation
DecisionNode		$a \in \mathbf{N}_{\text{all}} \Rightarrow$ $G(a \rightarrow \bigvee_{i=1, \dots, n} F b_i)$ \wedge $\bigwedge_{i=1, \dots, n} (b_i \in \mathbf{N}_{\text{all}} \Rightarrow$ $G(b_i \rightarrow O a))$
MergeNode		$\bigwedge_{i=1, \dots, n} (a_i \in \mathbf{N}_{\text{all}} \Rightarrow$ $G(a_i \rightarrow F b))$ \wedge $b \in \mathbf{N}_{\text{all}} \Rightarrow$ $G(b \rightarrow \bigvee_{i=1 \dots n} O a_i)$
ForkNode		$a \in \mathbf{N}_{\text{all}} \Rightarrow$ $G(a \rightarrow \bigwedge_{i=1, \dots, n} F b_i)$ \wedge $\bigwedge_{i=1, \dots, n} (b_i \in \mathbf{N}_{\text{all}} \Rightarrow$ $G(b_i \rightarrow O a))$
JoinNode		$G(\bigwedge_{a_i \in \mathbf{N}_{\text{all}}} F a_i \rightarrow F b)$ \wedge $b \in \mathbf{N}_{\text{all}} \Rightarrow$ $G(b \rightarrow \bigwedge_{i=1 \dots n} O a_i)$

expresses that each execution of a_i (for $i = 1, \dots, n$) is eventually followed by an execution of b . The second case expresses that each execution of b is preceded by at least one execution of the Action a_i (for $i = 1, \dots, n$). This results in the LTL formula $G(b \rightarrow \bigvee_{i=1 \dots n} O a_i)$.

As an example consider again process constraint #3. First of all, there is an AfterEdge from the Action “receive payment” to the Action “report payment” resulting in formula 7. The Actions “report rejected order” and “report payment” are connected to the MergeNode with AfterEdges. The MergeNode is connected to “close bill” via an AfterEdge and the Action “close bill” is an AllAction. Using our translation the following LTL formula is obtained:

$$G(\text{receive_payment} \rightarrow F \text{report_payment}) \wedge \quad (7)$$

$$G(\text{close_bill} \rightarrow$$

$$(O \text{report_payment} \vee O \text{report_rejected_order})) \quad (8)$$

The LTL formula 8 specifies that whenever the Action close_bill is executed, it has to be preceded by the execution of the Action report_payment or by the execution of the

Action $\text{report_rejected_order}$.

ForkNodes Let $f \in \mathbf{N}_f$ and $a \in \mathbf{N}_{\text{all}}$ and $e(a, f) \in \mathbf{E}$ and $\forall i = 1, \dots, n : e_i(f, b_i)$ and $b_i \in \mathbf{N}$, this results in the LTL formula $G(a \rightarrow \bigwedge_{i=1, \dots, n} F b_i)$. This formula expresses that on each path where a is executed, this execution has eventually to be followed by the execution of all the b_i actions ($\forall i = 1, \dots, n$). If at least one b_i is an AllAction, this results in the LTL formula $\bigwedge_{b_i \in \mathbf{N}_{\text{all}}} G(b_i \rightarrow O a)$.

As an example consider again process constraint #4. First of all, there is an AfterEdge from the Action “fill order” to the ForkNode. The ForkNode has two outgoing AfterEdges that connect the ControlNode to the Action “ship” and “send invoice” resp. . Using our translation the following LTL formula is obtained:

$$G(\text{fill_order} \rightarrow (F \text{ship} \wedge F \text{send_invoice})) \quad (9)$$

The LTL formula (9) specifies that whenever the Action fill_order is executed, it has to be eventually followed by the execution of the ship Action and by the execution of the send_invoice Action.

JoinNodes Consider again $j \in \mathbf{N}_j$ and $b \in \mathbf{N}_{\text{all}}$ and $e(j, b) \in \mathbf{E}$ and $\forall i = 1, \dots, n : e_i(a_i, j)$ and $a_i \in \mathbf{N}$.

This results in the LTL formula $G(\bigwedge_{i=1,\dots,n} O a_i)$ expressing that if Action b is executed it has to be preceded by the execution of all Actions a_i , where $a_i \in i = 1, \dots, n$. Each $a_i \in N_{\text{all}}$ results in the LTL formula $G(F a_i \rightarrow F b)$ expressing that after each $a_i \in N_{\text{all}}$ has been executed, then also Action b has to be eventually executed.

4.3 Tool Chain

We have set up a tool chain for the verification process (cf. Fig. 4) of process patterns in business processes. Therefore, we have developed an integrated workbench as an Eclipse plugin. Figure 7 shows a typical situation. On the left hand side, different business processes and patterns can be organized in projects. In the upper part, a business process is being modeled using the build-in Activity Diagram editor. In the middle part, process patterns can be modeled using the PPSL. Triggered by user interaction, the conformance of the business process with selected process patterns can be checked automatically. The result of the model checker is presented in the lower part of the workbench. The layout of the different editors and views of the workbench can be customized by the user, as typical for the eclipse workbench.

When the user triggers the verification, the complete tool chain of Fig. 4 is enacted automatically. The transition system generated by GROOVE is automatically translated into the input language of the NuSMV model checker [2]. The selected process pattern is automatically translated into temporal logic formulas as described in the previous section. Finally, the model checker is started with the transition system and the temporal logic formulas as input.

A future version is intended to allow for visual back-annotation of the result of the model checker in the pattern editor. Also, at a later stage, the system is intended to also interactively assist a process developer in correctly implementing process patterns into existing processes that do not yet fulfill the requirements given by a process pattern.

The implementation of the verification process is written in a modular way. Translating business process models and translating the process patterns are independent activities as well as the checking of the patterns. Therefore, single tools can be exchanged unproblematically.

5 Discussion and Conclusion

In this paper, we have introduced an approach to automatically check process constraints and demonstrated the application for checking quality constraints in business processes. In our approach, such process constraints are formally described through process patterns based on UML Activities. These patterns are the basis for checking business processes for conformance with the respective process

constraints. For this, the process patterns are transformed into temporal logic while the business process is transformed into a transition system. Together, this enables the application of model checking for ensuring conformance of the business process to the patterns defining the required process constraints. Thus, this technique allows formal verification of process constraints in business processes.

Furthermore, we have introduced tool support for defining and verifying such constraints by means of an Eclipse plugin. In a current project, this tool will be used to verify large-scale industry processes from the banking sector. Increasing “industrialization” in the finance business leads to the demand for well-defined business processes that interact seamlessly. Therefore, many requirements related to the processes have to be defined and verified. This will also be the basis to further investigate whether additional PPSL model elements and corresponding semantics are necessary to be able to express all typical sorts of constraints that occur in practice.

There are some more issues that need to be investigated. Different patterns can depend on each other or even contradict one another. The knowledge of these interdependencies between patterns can be used in tool support to increase the efficiency of the pattern checking process. Finally, we will also investigate how the occurrence of a process pattern can be located in a business process model.

References

- [1] S. W. Ambler. *Process Patterns - Building Large-Scale Systems Using Object Technology*. SIGS Books/Cambridge University Press, Cambridge, 1998.
- [2] A. Cimatti, E. Clarke, F. Giunchiglia, and M. Roveri. NuSMV: a new symbolic model verifier. In N. Halbwegs and D. Peled, editors, *Proceeding of International Conference on Computer-Aided Verification (CAV'99)*, volume 1633 of LNCS, pages 495–499. Springer, July 1999.
- [3] B. Devereux and M. Chechik. Automated support for building behavioral models of event-driven systems. In *Fundamental Approaches to Software Engineering (FASE)*, volume 3922 of LNCS, pages 122–138. Springer, 2006.
- [4] G. Engels, J. H. Hausmann, R. Heckel, and S. Sauer. Dynamic meta modeling: A graphical approach to the operational semantics of behavioral diagrams in uml. In A. Evans, S. Kent, and B. Selic, editors, *UML 2000, York, UK, October 2-6, 2000, Proceedings*, volume 1939 of LNCS, pages 323–337. Springer, 2000.
- [5] R. Eshuis. Symbolic model checking of uml activity diagrams. *ACM Trans. Softw. Eng. Methodol.*, 15(1):1–38, 2006.
- [6] A. Förster and G. Engels. Quality ensuring development of software processes. In F. Oquendo, editor, *Software Process Technology, 9th European Workshop, EWSPT 2003*, volume 2786 of LNCS, pages 62–73. Springer, 2003.

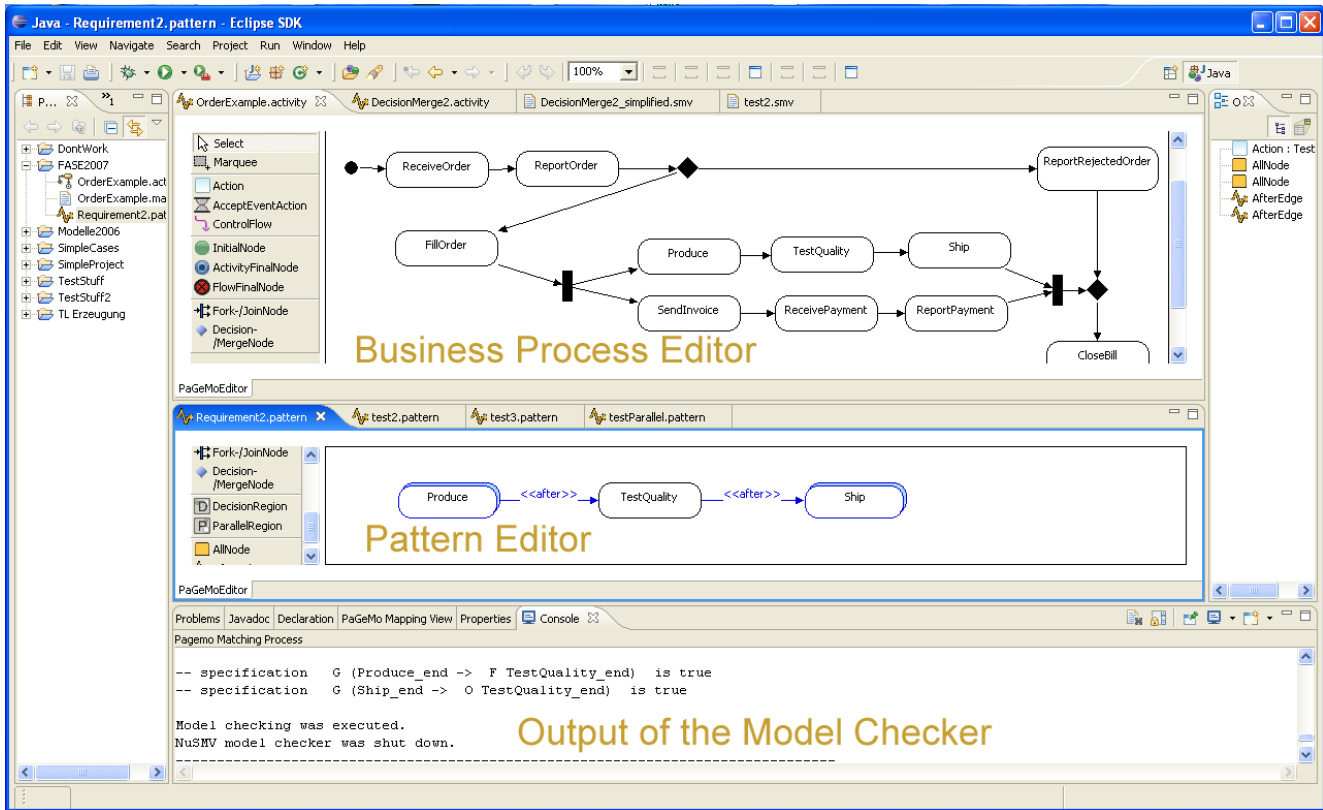


Figure 7. Eclipse plugin for modeling and checking quality constraint patterns

- [7] A. Förster, G. Engels, and T. Schattkowsky. Activity diagram patterns for modeling quality constraints in business processes. In L. C. Briand and C. Williams, editors, *Model Driven Engineering Languages and Systems, 8th International Conference*, volume 3713 of *LNCS*, pages 2–16. Springer, 2005.
- [8] A. Förster, G. Engels, T. Schattkowsky, and R. V. D. Straeten. A pattern-driven development process for quality standard-conform business process models. In *IEEE Symposium on Visual Languages and Human-Centric Computing VL/HCC 2006*, 2006.
- [9] D. M. Gabbay. The declarative past and imperative future: Executable temporal logic for interactive systems. In *Temporal Logic in Specification*, pages 409–448, 1987.
- [10] J. H. Hausmann. *Dynamic Meta Modeling. A Semantics Description Technique for Visual Modeling Languages*. PhD thesis, Universität Paderborn, Germany, 2005.
- [11] J. H. Hausmann, R. Heckel, and S. Sauer. Towards dynamic meta modeling of uml extensions: An extensible semantics for uml sequence diagrams. In *2002 IEEE CS International Symposium on Human-Centric Computing Languages and Environments (HCC 2001), September 5-7, 2001 Stresa, Italy*, pages 80–87. IEEE Computer Society, 2001.
- [12] W. Janssen, R. Mateescu, S. Mauw, P. Fennema, and P. van der Stappen. Model checking for managers. In *Proceedings of the 5th and 6th International SPIN Workshops on Theoretical and Practical Aspects of SPIN Model Checking*, pages 92–107, London, UK, 1999. Springer-Verlag.
- [13] E. Kindler and W. M. P. van der Aalst. Liveness, fairness, and recurrence. *Information Processing Letters*, 70(6):269–274, 1999.
- [14] Object Management Group. Unified Modeling Language 2.0 Superstructure Specification. <http://www.omg.org/cgi-bin/apps/doc?formal/05-07-04.pdf>, February 2006.
- [15] A. Rensink. The GROOVE simulator: A tool for state space generation. In J. Pfalz, M. Nagl, and B. Böhlen, editors, *Applications of Graph Transformations with Industrial Relevance (AGTIVE)*, volume 3062 of *LNCS*, pages 479–485. Springer, 2004.
- [16] D. Riehle and H. Zullighoven. Understanding and using patterns in software development. *Theory and Practice of Object Systems*, 2(1):3–13, 1996.
- [17] H. Störrle and J. H. Hausmann. Towards a formal semantics of uml 2.0 activities. In P. Liggesmeyer, K. Pohl, and M. Goedicke, editors, *Software Engineering*, volume 64 of *LNI*, pages 117–128. GI, 2005.
- [18] W. van der Aalst, A. ter Hofstede, B. Kiepuszewski, and A. Barros. Workflow patterns. In *Distributed and Parallel Databases*, volume 14(3), pages 5–51, July 2003.