Volker Gruhn, Ralf Laue, Stefan Kühne, Heiko Kern

A Business Process Modelling Tool with Continuous Validation Support

In this article, we want to present the prototype of a modelling tool that applies graph-based rules for identifying problems in business process models. The advantage of our approach is twofold. Firstly, it is unnecessary to compute the complete state space of the model in order to find errors. Secondly, our technique can be applied to incomplete business process models. Thus, the modeller can be supported by direct feedback during the model construction. This feedback does not only report problems, but it also identifies their reasons and makes suggestions for improvement.

1 Introduction

Validation of business process models has been studied for a long time. In a recent paper, Wynn et al. (2009) write that 'process verification has matured to a level where it can be used in practice'. Although this is good news, we argue that many of the current approaches do not yet support the business process modeller in an optimal way. The reason for this argument is that most validation methods are applied only after the model has already been completed. For example, most methods which transform a business process model into an analysable Petri net have problems with incomplete models.

In this article, we present a validation approach that gives the modeller an immediate feedback about modelling errors. A prototypical implementation of our approach has been integrated into the business process modelling editor *bflow* (www.bflow.org). It locates not only 'technical' errors (such as deadlocks in the control flow), but also parts of the model that can be regarded as 'bad style'. The modeller not only receives the information with which the model has problems, but our tool also shows the locations of error causes in the visual representation and suggests how to fix these problems. A rule language allows the users to add their own rules, for example, rules for checking company-wide style guidelines.

The principle behind our approach is called 'continuous validation'. It can be compared to techniques such as continuous compilation and continuous testing, which are integrated into modern software development systems. In our approach, continuous validation can help to detect and fix errors at a early stage in process modelling.

2 Basic Concepts and Definitions

2.1 Event-Driven Process Chains

There exist several languages for graphical business process modelling. In this paper, we use Event-Driven Process Chains (EPC, van der Aalst, 1999) to demonstrate our approach. However, the underlying principles can also be applied to other languages such as BPMN (as shown in Laue and Awad, 2009).

We would like to start with a semi-formal description based on the meta model given in Fig. 1. EPC models are finite directed coherent graphs consisting of non-empty sets of nodes and arcs. Nodes are either functions (activities which need to be executed, depicted as rounded boxes), or events (representing pre- and postconditions of functions, depicted as hexagons) or connectors. Arcs between these elements represent the control flow. A function has exactly one incoming and exactly

one outgoing arc. An event has at most one incoming and at most one outgoing arc. An event without incoming arcs is called a 'start event', and an event without outgoing arcs is called an 'end event'.

The connectors are used to model parallel and alternative executions. There are two kinds of connectors – splits and joins. Splits have one incoming and at least two outgoing arcs, joins have at least two incoming arcs and one outgoing arc.

AND-connectors (depicted as ⓐ) are used to model parallel executions. When an AND-split is executed, the elements on all outgoing arcs have to be executed in parallel. An AND-join connector waits until all parallel control flows on its incoming arcs are finished. XOR-connectors (depicted as ⓧ) can be used to model alternative executions: An XOR-split has multiple outgoing arcs, but only one of them will be processed. An XOR-join waits for the completion of the control flow of one of its incoming arcs. If a flow arrives from more than one arc, most definitions for a formal EPC semantics regard it as a synchronisation error. The control flow is not forwarded in this case. OR-connectors (depicted as ⓥ) are used to model parallel executions of one or more flows. An OR-split starts the processing of one or more of its outgoing arcs. That is, after an OR-split with n outgoing arcs, at least one of those arcs and at most all n arcs become active. An OR-join waits until all control flows that can reach this join are finished.

Fig. 2 shows a simple order process modelled as EPC. Note that splits and joins neither necessarily occur pairwise nor form necessarily a well-structured model. In fact, the notation allows arbitrary combinations of connectors which are often the cause of modelling errors.

A state of an EPC is a binary marking of its elements, i.e., some elements of an EPC are marked as *active* by placing tokens on them. A state is a start state if only start events are marked. A sequence of states is an execution of the business process model. Its semantics is defined by a transition relation, i.e., a set of rules that define under which circumstances a state S_1 in this sequence is allowed to be followed by a subsequent state S_2. Several different definitions exist for transition relations but because of space restrictions we will omit a detailed discussion. The interested reader is referred to Kindler (2004); Wynn (2006); Mendling (2007).

2.2 Control Flow Errors

The 'soundness' property is the primary correctness criterion for business process models. It has been defined by van der Aalst (1999) by the following three properties:

1. In every state that is reachable from a start state, there must be the possibility to reach a final state, i.e., a state without a subsequent state according to the transition relation '(option to complete)'.
2. If a state has no subsequent state (according to the transition relation that defines the precise semantics), then only end events must be marked in this state '(proper completion)'.
3. There is no element of the EPC that is never marked in any execution of the EPC '(no needless elements)'.

Violations of the soundness criterion usually indicate an error in the model. A typical example is a deadlock situation with an XOR-split whose outgoing arcs are later joined by an AND-join. This example would lead to a violation of the second property: It is possible that no further progress in the execution of the EPC can be made, but the elements at the incoming arcs of the AND-join are still marked because the AND-join has to wait until 'all' incoming arcs have been traversed.

3 Existing Validation Methods

Lindland et al. (1994) point out that 'modelling is essentially making statements in some language'.

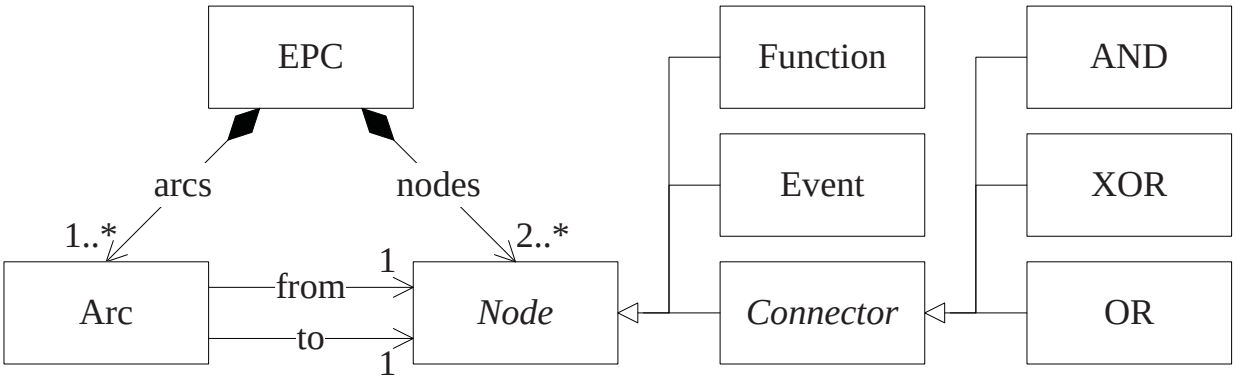


Figure 1: EPC metamodel

For this reason, they use linguistic concepts for assessing the quality of models. They propose a framework that distinguishes between three quality goals:

- 'Syntactic Quality', i.e., how far the model adheres to the syntactical rules of the modelling language
- 'Semantic Quality' which includes validity (all statements made in the model are correct) and completeness (the model contains all statements about the domain that are relevant)
- 'Pragmatic Quality' which addresses the requirement to chose the best (i.e., most comprehensible) way to express some meaning

In this chapter we discuss existing validation methods and available tools for each of these quality goals.

3.1 Syntactic Quality

The syntax of a graphical model defines in which way the modelling elements can be arranged in order to build a valid model. For EPC models, the syntax is mainly described by the metamodel shown in Fig. 1. However, some syntactical restrictions are not captured by the metamodel, for example that it is forbidden to have a cycle in the model which consists only of connectors. Therefore, validating the syntactical correctness of an EPC is more than assuring its compliance with the metamodel.

In Mendling and Nüttgens (2003), Mendling and Nüttgens have evaluated the ability of several XML tools to check the semantic correctness of EPC models. They came to the result that the language 'Schematron' can check the most, but not all syntactic requirements. In particular, this language cannot validate although the graph is strongly connected, that no cycles are formed by connectors only and that for each modelling element e there is a path from a start event to e, as well as from e to an end event. After the publication of Mendling and Nüttgens in 2003, more sophisticated XML checking tools have been developed that can be used to validate all syntactic requirements for EPC models, namely Incox (Opočenská and Kopecký, 2008) and the Constraint Language in XML (CliCML Jungo et al., 2006). By adding the syntactic requirements as OCL constraints to the metamodel, it is also possible to use OCL tools for validating the syntactic correctness of a model.

In Gruhn and Laue (2007a), it was shown that the rule based approach that is used for validation in *bflow* is able to validate all semantic requirements for EPC models. We see an advantage of this approach in the fact that syntactic quality is

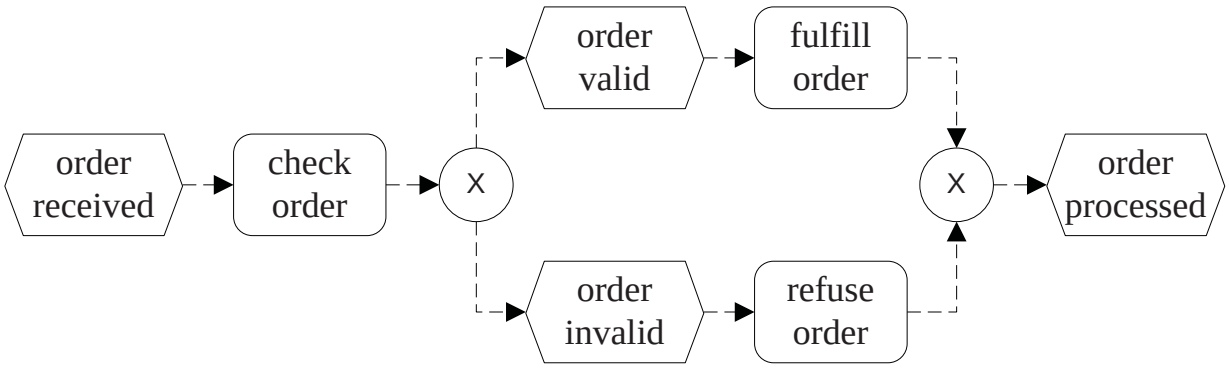


Figure 2: An example EPC

validated using the same approach that can is applied for reasoning about semantic and pragmatic quality.

3.2 Semantic Quality

The problem to overcome when verifying the semantic quality of business process models (like EPCs) is that the modelling language has been introduced 'without defining their semantics'. For this reason, the first step in a verification process is usually to transform the model into a formalism with well-defined semantics. In fact, the rules for such a transformation 'define' the semantics of the model. Petri nets are the natural choice for that purpose. They have been used by several authors (van der Aalst, 1999; van Dongen et al., 2005; Mendling, 2007)[1]. Other formalisms include abstract state machines (Eshuis, 2002) and Pi calculus (Puhlmann, 2007).

Once an EPC model has been transformed into a Petri net, the whole range of existing tools for analysing Petri nets is available, as introduced in Langner et al. (1997); van Dongen et al. (2007); Wynn (2006); Mendling (2007). The Petri net based approach works in practice, but has two disadvantages. First, often the analysis result covers only the information whether the model contains errors, without giving feedback about the reason for an error. Even if the verification tool *translates* a counterexample from the Petri net back to the EPC model, it can happen that information will be lost. For example, Fig. 3 obviously contains two synchronisation problems – one in the outer AND-XOR control block (s_1, j_1) and another one in the inner AND-XOR control block (s_2, j_2). However, the synchronisation error in the inner control block implicates that the execution always blocks, and there will never be an execution where both incoming arcs of the rightmost XOR-join are enabled. For this reason, a dynamic analysis tool that explores the state space will be unable to report the problem of the outer AND-XOR control block. The second disadvantage is that it is often impossible to locate errors in models that are not yet completed (e.g., EPCs containing several sub-graphs which are not yet connected with each other).

Another well-studied method for the validation of EPCs and similar models is the application of reduction rules (cf. e.g., Sadiq and Orlowska, 2000). The idea of the reduction approach is to delete repeatedly sections from an EPC which are well-structured (for example, a control flow block where an AND-split is matched by an AND-join) and are thus trivially correct. If an EPC can be reduced to a single node in this way, it is correct. Otherwise, no answer about its correctness can be given. That is, the answer to the question 'Are there any problems with the model?' is either 'No' or 'Unable to decide', which is far from

[1] The given references are not exhaustive. A more detailed categorization of related work can be found in van Dongen et al. (2007), Mendling (2007) and Morimoto (2008).

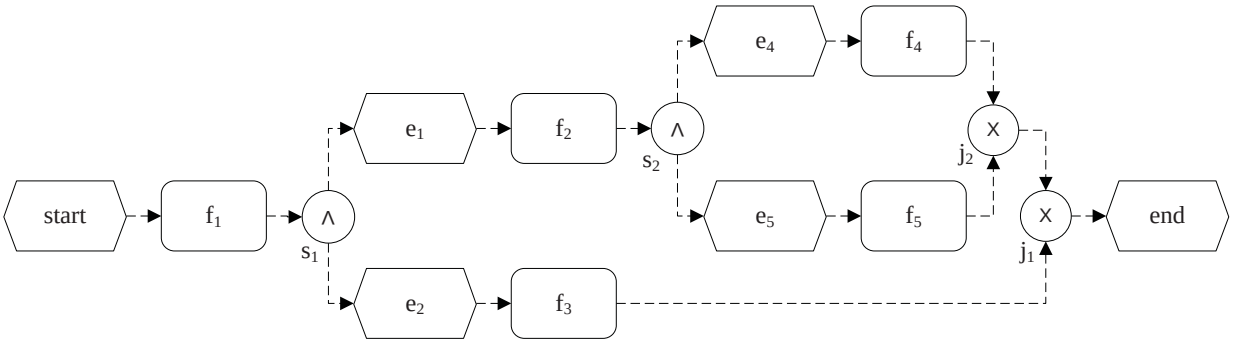


Figure 3: Nested mismatched AND-splits and XOR-joins

the desired 'No' or 'Yes, and the problems are as follows...'. However, recent work by Mendling (2007) has made a fruitful contribution. By considering typical error situations in the reduction rules, Mendling's approach allows us to answer our question about errors in the model with 'No', 'Yes, and the problems are...' or 'Unable to decide'. In the latter case, Petri net based methods can still be used to come at least to a 'Yes or No' answer.

Our work has also been influenced by the approach described in van Dongen et al. (2006). This approach adapts ideas from Petri net theory like the concept of handles (Esparza and Silva, 1989; van der Aalst, 1998) to EPCs. In van Dongen et al. (2006), the authors locate causal relationships between parts of an EPC. These relationships (called 'causal footprints' in van Dongen et al., 2006) are relations like 'after element x has been processed, at least one of the elements in the set $\{y, z\}$ has to be processed'. Using this method, error patterns can be detected and the reasons for errors can be identified. Moreover, the method works on incomplete models as well. Unfortunately, in the form as described in van Dongen et al. (2006), the approach has not yet matured for practical use. A minor reason is that it does not work with EPCs containing OR-connectors (but it would not be too much effort to expand the method such that OR-connectors could be considered). But more important is that the computation of relationships among elements is far too slow because too many relations have to be considered.

Another promising approach has been described in Vanhatalo et al. (2007, 2008). It uses decomposition of workflow graphs into single-entry-single-exit-fragments and can quickly classify some fragments as sound or unsound. A validation in Vanhatalo et al. (2007) shows that the approach is suitable for the majority of a sample of industrial business processes.

3.3 Pragmatic Quality

Pragmatic model quality refers to the best choice among several possible ways to express the same meaning. Current business process modelling tools provide limited support for improving pragmatic quality. The only implementation we are aware of is a validation built into the *YAWL Editor* (Wynn, 2006) which allows to find OR-joins that should be replaced by AND-joins or XOR-joins in order to use the modelling element that expresses the meaning of the model in the most comprehensible way.

4 Immediate Validation Feedback in Business Process Modelling

4.1 Validation Approach

From our point of view, a sophisticated validation support of business processes should give immediate and continuous feedback to business analysts about weaknesses and inconsistencies in possibly incomplete models. The established modelling

process with sequential modelling, validation and evolution stages should be shortened as far as possible to a modelling process with integrated validation support.

We took an inspiration from existing programming environments which offer continuous syntax checking. Static analysis tools such as FindBugs (Hovemeyer and Pugh, 2004) can also find patterns in the code that relate to possible errors. Fault information and warnings can be presented to the developers while they are typing (Layman et al., 2008).

Most current modelling tools still lack this kind of support. A remarkable exception is the UML modelling tool ARGO/UML (Robbins and Redmiles, 2000), which runs some checks (called design critics) in the background. While the architecture of ARGO/UML allows the user to add own design critics, the vast majority of currently implemented checks remain on a syntactical level. In contrast, we would like to direct the attention not only to problems on a syntactical level but also to semantic and even pragmatic issues.

Furthermore, we would like to note a difference between our approach and existing tools like *IBM MQSeries Workflow* and *AristaFlow* which impose well-structuredness rules on models and allow to create well-structured models only (which are always sound). We do not want to restrict the modeller, but give him or her feedback about possible errors and improvements.

Our intended validation support is based on the following principles:

1. For adaptability and extensibility reasons, the validation rules should be expressed in a modular and human readable manner. For this purpose, we propose declarative validation rules which enable the expression of additional error patterns and modelling idioms by adding new rules without effecting existing ones.
2. To avoid disadvantages of non localised error messages, the validation strategy should work on input models in a native way. The validation rules should refer to model fragments and identify syntactic structures that may cause errors during runtime.
3. The validation rules should be expressed fine-grained enough to produce meaningful error feedback.
4. Recurring model navigations and computations of model properties should be defined as reusable functions that are used in validation rules. The set of helper functions represents an extensible library that eases the definition of validation rules in terms of an (internal) domain-specific language.
5. A rather soft requirement is that the validation solution should be seamlessly integrated into the modelling tool of choice with the ability to annotate error causes and to suggest possible improvements.

The instantiation of these principles results in process-specific validation rules and process-specific helper functions. Before we describe them more detailed, we demonstrate how the validation approach is implemented.

4.2 Declarative rules with Prolog

As described in Gruhn and Laue (2007a), the information that is included in an EPC diagram can be translated into logic facts, expressed in the logic programming language Prolog.

Listing 1: Prolog facts generated from an EPC

```
event(i_1).
bflow_id(i\_1, '\_nElA8C2fEd6OHrn24qlDpw').
elementname(i_1,'Order received').
function(i\_2).
bflow_id(i\_2, '\_aLM6wC2fEd6OHrn24qlDpw').
elementname(i\_2,'check order').
xor(i\_3).
bflow_id(i\_3, '\_kVfkgC2fEd6OHrn24qlDpw').
arc(i\_1,i\_2).
arc(i\_2,i\_3).
```

As an example, the three leftmost nodes in Fig. 2 and the arcs between them are represented by the facts shown in Listing 1. The clauses named `bflow_id` express the relationship between the id's used for the nodes in the Prolog program and the id's assigned by EMF. This allows to provide feedback directly related to the diagram elements where the problems occur.

A typical Prolog rule that checks for modelling problems is shown in Listing 2. If an AND-split or OR-split is found to be an exit from a loop, this is reported as an error (because in such a situation the loop could be executed infinitely often). The clause `cycle_exit` (defining when a node is called an exit from a loop) has been defined before and can be used within the rule. Because of this modular design, it is not difficult to add new rules.

Listing 2: Prolog rule to find loops ended by something else than an XOR split

```
loop_exit_error(X) :– (andsplit(X);orsplit(X)),
      cycle_exit(X),
message('ERROR',X,'This connector ends a loop; it
      should be an XOR split'),!.
```

4.3 Graph-based pattern matchings with Check and XTend

The declarative validation approach based on Prolog rules enables the modular and intuitive expression of validation rules for business process models. Because of the underlying generic unification algorithm the facilities to control performance issues are limited. To support advanced rules with complex graph computations we use a hybrid approach provided by the framework openArchitectureWare (oAW)[2].

Validation rules following this approach are expressed in the oAW Check language (see e.g., Listing 3). A declarative rule is introduced by a context specifying a metamodel element which instances are validated (e.g., epc::Connector). The set of model elements can optionally be restricted by an if-clause. The keyword **ERROR** signals that a violated validation rule represents an error. A corresponding advice is specified. **WARNING** provides an alternative feedback category. The Boolean expression after the colon specifies a validation assertion which holds for valid models.

Listing 3: A check rule

```
// A join connector is is not also a split
context epc::Connector
    if this.isJoin()
    ERROR \emph{Connector is a split and a join as
        well. Only one of them is allowed.}:
    !this.isSplit();
```

The example given in Listing 3 assures that AND-, OR- or XOR-connectors do not have both more than one incoming arc and more than one outgoing arc. The restriction expression as well as the assertion expression refer to separately defined helper functions implemented in the functional programming language oAW XTend. XTend is designed for model to model transformations. In our case, XTend is used to express queries on the input model to calculate properties of model elements.

Listing 4: An XTend function

```
// Is a connector a join–connector
cached Boolean isJoin(epc::Connector c) :
    c.incomingArcs().size > 1;
```

The definition of the function isJoin() used in Listing 3 is shown in Listing 4. The type of the return value is set to Boolean. The function takes a single argument c of type epc::Connector. The keyword **cached** signals that the result of the function call is stored. A second function call with the same argument results in the stored return value, i.e., the function is executed only once. The function isJoin() relies on the incomingArcs() function, which returns the set of arcs flowing into the given argument. Similar to the function calls in the validation rule (see Fig. 3), the function call

[2] http://www.eclipse.org/gmt/oaw/

is expressed with the help of the concatenation operator .. The left hand side of this operator is handled as the first argument of the right hand side which allows the definition of readable expressions.

Based on basic helper functions we introduced specific classes of functions which eases pattern matchings. To validate well-structured EPC fragments, corresponding split and join pairs need to be identified. For this purpose, we define the concept *match* as follows:

Definition 1 A split s is matched by a join j (*matches*(s, j)) if there exist two directed paths from s to j whose only common elements are s and j.

A necessary condition for a matching split/join pair is that the split is connected to the join. Considering the set A of arcs between the split and the join, the existence of two different paths assures that there is no single arc α which divides A into two disjunctive subsets of preceding arcs of α and succeeding arcs of α. We call this concept *separation*. The corresponding XTend function is given in Listing 5.

Listing 5: The function separates()

```
cached Boolean separates(epc::Arc arc, epc::
    Connector source, epc::Connector target) :
   source.arcsBetween(target)
       .remove(arc)
       .without(arc.from.precedingArcs({source}.
           toSet()))
       .without(arc.to.succeedingArcs({target}.toSet
           ()))
       .isEmpty;
```

With the help of the function separates() the match property of a given split s and a join j can be derived. If s has two children n_1 and n_2 that are connected to j and there is a separating arc α with *separates*(α, s, j) then the paths from n_1 to j and n_2 to j share a common join j_c with an outgoing arc α_c which also separates s and j. Hence, the decision of the match property (*matches*(s, j)) depends on outgoing arcs of joins between s and j and the search space of separating arcs can be restricted accordingly.

Listing 6: The function matches()

```
cached Boolean matches(epc::Connector split, epc::
    Connector join) :
   split != join
   && split.children().select(c| c == join || c.
       isConnectedTo(join)).size > 1
   && split.joinsBetween(join).outoutgoingArcs()
       .notExists(a| a.separates(split, join));
```

The EPC in Fig. 3 contains two splits s_i and two joins j_i. Obviously, s_1 matches j_1 and s_2 matches j_2. s_1 does not match j_2 because the child e_1 of s_1 is not connected to j_2. s_2 does not match j_1 because the outgoing arc of j_2 separates the arcs between s_2 and j_1.

The two matching connector pairs (s_1, j_1) and (s_2, j_2) in Fig. 3 share another essential property. They delimit model fragments with a single entry and a single exit (called SESE fragments), i.e., there are no *entries into* or *exits from* the fragments. The terms *entry* and *exit* are defined as follows:

Definition 2 Let s be a split and j be a join. We say that there are no entries and no exits between s and j (*sese*(s, j)) if the following conditions hold:

1. Every path from s to an end event contains j.
2. Every path from a start event to j contains s.
3. Every path from s to s contains j.
4. Every path from j to j contains s.

The properties of a SESE fragment exclude additional entries and exits to the fragment. As a consequence of this, the succeeding arcs of the entry bordered by the exit correspond to the preceding arcs of the exit bordered by the entry. A comparison of these two sets of arcs leads to an efficient identification of SESE fragments (see Listing 7).

Listing 7: The function sese()

```
cached Boolean sese(epc::Node source, epc::Node
      target) :
    source.succeedingArcs({target}.toSet())==target.
          precedingArcs({source}.toSet());
```

With the help of the functions matches() and sese() SESE matches can be identified as conjunction of these two properties (see Listing 8).

Listing 8: The function seseMatch()

```
cached Boolean seseMatches(epc::Node source, epc::
      Node target) :
    source.sese(target) && source.matches(target);
```

The identification of SESE fragments can be extended to fragments with additional entries and exits. For this, the set of border elements of the succeeding arc computation or the preceding arc computation respectively has to be extended. Considering an additional border element at the succeeding arc computation, for instance, allows the identification of fragments with a single entry and two exits.

The conjunction of these kinds of structures and the match property leads to the following concepts of blocks between a split s and a join j for which *matches*(s, j) hold:

- **match with exit** (if there is a path from s to an end event that does not pass j or if there is a path from s to s which does not pass j)
- **match with upstream entry** (if there is a path from a start event to j which does not pass s)
- **match with downstream entry** (if there is a path from j to j which does not pass s)

By analysing the possible combination of the types of splits and joins and possible entries into or exits from the block between split and join, we identified the cases that can lead to an error in the model.

4.4 Quick fixes with ECL

The validation approaches mentioned above are able to highlight weaknesses and inconsistencies in possibly incomplete models. The ability to suggest possible solutions for errors and warnings represents a further dimension of modelling support. We address this aspect with the Epsilon Validation language (EVL) which is part of the Epsilon management framework.[3] EVL provides the **fix** concept to specify an update transformation. Listing 9 shows an example that removes redundant connectors having only one incoming and outgoing control flow.

Listing 9: A rule with quick fix

```
context Element {
    constraint RedundantConnector {
        guard : self.isConnector()
        check : not (self.out.size = 1 and self.\emph{
            in}.size = 1)
        message : 'Connector is redundant.'
        fix { title : 'Delete redundant connector.'
            do {
                var newArc := new Arc;
                var epc : Epc := self.eContainer();
                var incomingArc := self.\emph{in}.
                    first();
                var outgoingArc := self.out.first();

                newArc.from := incomingArc.from;
                newArc.to := outgoingArc.to;
                epc.elements.remove(self);
                epc.connections.add(newArc);
                epc.connections.remove(
                    incomingArc);
                epc.connections.remove(
                    outgoingArc); }
        }
    }
```

5 Examples

In this section, we discuss typical classes of modelling errors that are identified by checking rules

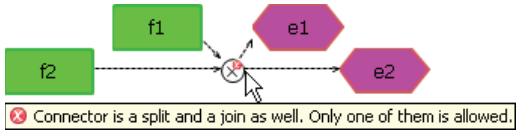
[3] http://www.eclipse.org/gmt/epsilon/

included in our tool.

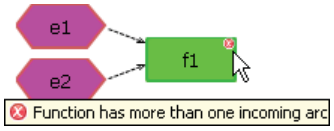
5.1 Syntax Errors

A less complicated task is to check an EPC (or to be precise: a construction that is supposed to be one) for syntactical correctness. The EPC metamodel given in Fig. 1 does not represent all syntactical requirements, e.g., the non-existence of cycles made from connectors (Nüttgens and Rump, 2002). Instead, the model space defined by the metamodel has to be restricted by some additional rules. Writing such syntax rules is a rather easy task (Gruhn and Laue, 2007a) and is supported in our approach.

Fig. 4(a) shows a syntax error which violates the validation rule given in Listing 3. Another syntax error is that an event or a function has more than one incoming or outgoing arcs (see e.g., Fig. 4(b)). Such errors are not uncommon: We found 14 of them in the 604 models of the SAP reference model.



(a)



(b)

Figure 4: Connector with several incoming and outgoing arcs (a), function with several incoming arcs (b)

5.2 Control Flow Errors

Informally spoken, deadlocks and synchronisation errors in an EPC often result from conflicting connectors (for example, a combination between an XOR-join and an AND-split which will result in a deadlock).

The functions match() and seseMatch() allow us to find this type of control flow errors where the type of the split differs from the type of the join. From the 178 errors found by Mendling in the 604 models of the SAP reference model, 44 fell into this category (Mendling, 2007).

Fig. 5 shows an obvious error consisting of a mismatched XOR-split (the left one)/AND-join combination. The two XOR-connectors inside the fragment illustrate that the rule given in Listing 10 also finds errors in not well-structured models.

Listing 10: Mismatched XOR-split and AND-join

```
// XOR−AND−Mismatch
context epc::Connector if (this.isAndJoin())
    ERROR \emph{Mismatched XOR−split...}
    this.predessesors().notExists(p| p.isXorSplit() &&
        p.seseMatch(this));
```

For an XOR-split/OR-join combination, the situation is different: OR-joins synchronise all active incoming control flows. If there is only one such flow (as the result of the choice at the XOR-split), the execution continues without problems. For this reason, most verification approaches (remarkable exceptions are Wynn, 2006; Rump, 1999) do not complain about such combinations.

With our declarative validation approach based on graph patterns such situations can be handled similar to control flow errors. To produce improvement advises for mismatched XOR-split/OR-join connectors an additional *WARNING* rule similar to the *ERROR* rule given in Listing 10 has to be introduced with a changed restriction expression (OR-joins instead of AND-joins) and an adapted warning expression.

With the help of the XTend functions to identify fragments with multiple entries and multiple exits (see Sect. 4.3) similar rules can be defined.

Another subclass of rules is related to errors that occur in iterations (circles) in an EPC. Due to space restrictions, we do not describe them in detail.

For EPCs that occur in practice, the checks dis-

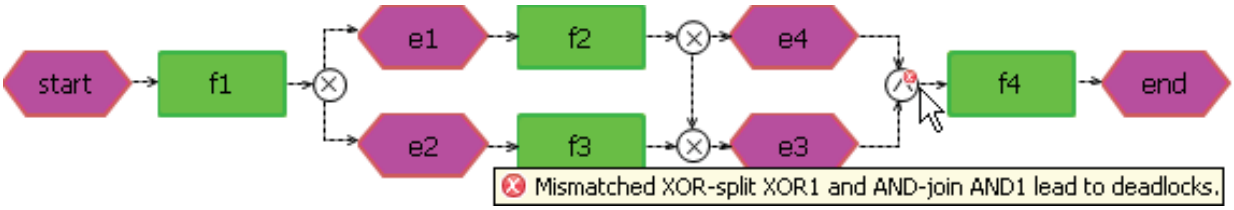


Figure 5: Mismatched XOR-split and AND-join

cussed in this section can be performed fast; running them repeatedly as a background task is not a problem.

5.3 Company-Wide Style Rules

The rule set in our tool is open for additional domain-specific rules. If, for instance, a company wants to transform EPC models directly into executable workflow models, it could be a requirement to use certain company-wide style rules for EPCs in order to disallow unstructured EPC models.
Such domain-specific adaptations can simply be realised in our tool by extending the predefined rule set. An example for such a rule is given in Fig. 6. The intention of the fragment between the two XOR-connectors is that function f_2 is executed only if event e_1 occurs which corresponds to a single if-expression in a programming language. However, because of a missing condition the arc connecting the two XOR-connectors might also be regarded as path that can always be executed. To avoid ambiguities a distinguishing event is desirable. With a rather simple check rule this pattern can be found and the suggestion to insert a negation of event e_1 can be created.

5.4 Pragmatic Problems

Instead of only detecting errors and violations of style rules, the approach described in this paper can also be used to locate possible pragmatic problems. In this case, the user is warned about the possible problem. It is the responsibility of the modeller to decide whether the model should actually be modified. An example for a situation where a warning is given is shown in Fig. 7. In this model fragment, a decision is followed by an XOR split, and one of two events occurs. However, for the further processing of the business process, it does not matter which of the events occurred. While in some cases, this can be a correct model anyway, the model in Fig. 7 indeed seems to contain an error: Most likely, the processing should stop if the customer is found not to be credit-worthy.

5.5 Detecting Problems in Textual Descriptions

Using the extended validation with Prolog rules, we are able to find possible problems that are related to the texts written in the labels of events and functions. A typical situation is shown in Fig. 8: If a function is labelled with a text like 'Decide whether x or y', it should most likely not be followed by an OR-split that would allow that both x and y take place. Currently, the Prolog rules implemented in the *bflow* Toolbox* contain seven checks that are related to the textual labels found in the model.

6 Validation

6.1 Correctness of our Pattern-Based Validation Approach

We searched for error patterns described above (and some more that cannot be described in detail due to space restrictions) in a repository of 984 EPC models.

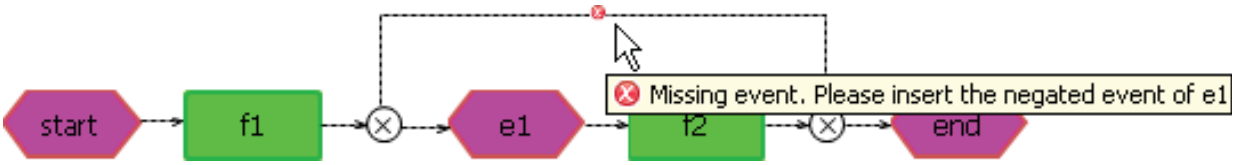


Figure 6: Missing event between an XOR-split and XOR-join with a suggestion for improvement

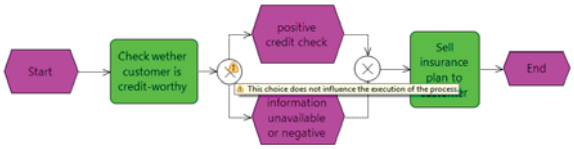


Figure 7: The upper event should lead to different activities than the lower one.

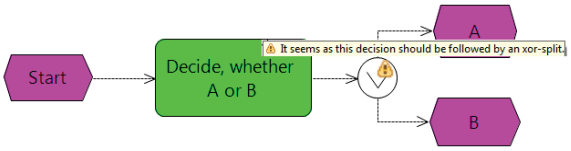


Figure 8: From the function's label, we can conclude that the OR should be replaced by an XOR.

Those models have been collected from 130 sources which can be categorized as follows:

- 531 models from the SAP R/3 reference model, a widespread business reference model
- 112 models from 31 bachelor and diploma theses
- 25 models from 7 PhD thesesses
- 13 models from 2 technical manuals
- 82 models from 48 published scientific papers
- 12 models from 6 university lecture notes
- 4 models from sample solutions to university examination questions
- 88 models from 11 real-world projects
- 88 models from 7 textbooks
- 29 models from 14 other sources

A Prolog version of the rules for checking semantics errors needed only 65 seconds for analysing 'all' models from the repository. This means that the checks are fast enough for being used at modelling time.

We also analysed the soundness property of all models using three well-known open source tools that check business process models for the soundness property: *EPCTools*, the *ProM plugin for EPC soundness analysis* and the *YAWL Editor*.

After doing the analysis with the different tools, we compared the results. Because of subtle differences among the tools when it comes to defining the semantics of the OR-join there were a few differences in the results of the tools. Details of those differences are discussed in Gruhn and Laue (2009a); here it is sufficient to say that in cases of differences among the tools we looked at the model and selected the result that complied with the intuitive understanding of its semantics.

The comparison between our heuristic results and the exact results showed that the heuristics worked almost as good as the state space exploring tools: For all models which have been categorized as not being sound by the *exact* tools, our Prolog program also found at least one pattern that (likely) shows a violation of the soundness property, i.e., we have had no *false negatives*. On the other hand, our program warned about a possible soundness violation for exactly one model that turned out to be sound, i.e., we have had only one *false positive*. It is worth mentioning that the model, for which this false positive occurred, was taken from Mendling et al. (2008) where it has been published as an example for bad modelling that should be improved.

All of the 'exact' tools failed to compute the soundness for some models. The reason is the state space explosion, i.e., the huge number of possible executions that have to be calculated. This state space explosion can be avoided by our pattern-based approach. More details about the validation can be found in Gruhn and Laue (2009b).

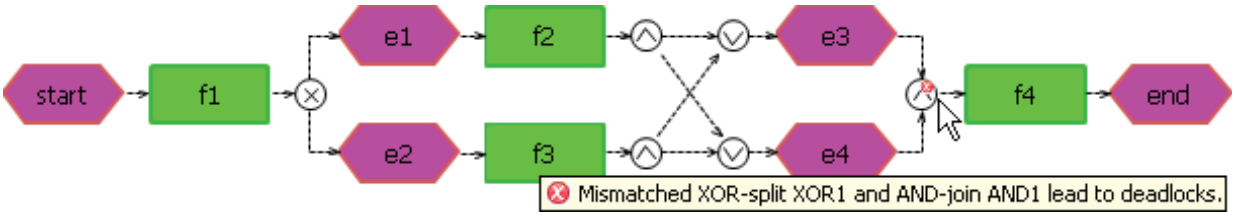


Figure 9: Sound EPC for which our tool would report a problem

6.2 Influence of Continuous Validation on Model Quality

To get a first impression on the effect that the continuous validation feature may have on the quality of models, we conducted an experiment with a group of business administration students at the the University of Applied Sciences Bonn-Rhein-Sieg, Germany. We gave the students the task to build a business process model from a case study.

One group of 7 students used the *bflow* Toolbox* with continuous validation, another group used the same tool where the continuous validation feature has been disabled. There were 6 syntactical errors in the 7 models of the group that used the validation feature, but 24 such errors in the 6 models of the other group. This result indicates that the presence of a continuous validation feature indeed can have a positive influence on model quality. Details of the experiments can be found in Laue et al. (2009). Further experiments with larger groups are planned to evaluate the effect of continuous validation more deeply.

7 Conclusions and Directions for Future Research

With the error patterns discussed in Sect. 5, we can already identify the vast majority of control flow problems in an EPC. However, the presented validation approach basically has a heuristic nature. The checks in the editor are not meant to be a complete validation. The EPC depicted in Fig. 9 shows two weaknesses of our approach. Although the model represents a sound EPC, the rules discussed so far produce a false error message for the AND-join. Furthermore our rules do not create hints according to the OR-joins which might be replaced by two XOR-joins. We deliberately did not try to include rules for such exotic cases.

An evaluation based on a set of 984 EPCs from various sources shows that a limited set of rules already detected almost all control-flow errors. We see the main advantage of our approach in the fact that information about possible problems in a model is immediately reported to the modeller, even before the model has been completed. These alerts provide suggestions for improvement and are given in a way that does not force the modeller's attention away from the modelling task.

It will be a direction of future research to deal with patterns where a model change could result in more *readable* models – even for EPCs where the original model did not have deadlocks and similar control flow problems (Gruhn and Laue, 2007b). We are also researching more problems that can be found by analysing the textual description of events and functions as discussed in Sect. 5.5.

References

van Dongen B, Mendling J, van der Aalst W (2006) Structural patterns for soundness of business process models. EDOC pp 116–128, DOI http://doi.ieeecomputersociety.org/10.1109/EDOC.2006.56

Eshuis R (2002) Semantics and verification of UML activity diagrams for workflow modelling. PhD thesis, University of Twente, Enschede

Esparza J, Silva M (1989) Circuits, handles, bridges and nets. In: Applications and Theory of Petri Nets, pp 210–242

Gruhn V, Laue R (2007a) Checking properties of business process models with logic programming. In: MSVVEIS 2007, INSTICC Press, pp 84–93

Gruhn V, Laue R (2007b) Good and bad excuses for unstructured business process models. In: Proceedings of 12th European Conference on Pattern Languages of Programs (EuroPLoP 2007)

Gruhn V, Laue R (2009a) A Comparison of Soundness Results Obtained by Different Approaches. In: 1st International Workshop on Empirical Research in Business Process Management

Gruhn V, Laue R (2009b) A heuristic method for business process model evaluation. In: Advances in Enterprise Engineering III, 5th International Workshop, CIAO! 2009, and 5th International Workshop, EOMAS 2009, held at CAiSE 2009, Amsterdam, The Netherlands, June 8-9, 2009. Proceedings, Springer, Lecture Notes in Business Information Processing, vol 34, pp 28–39

Hovemeyer D, Pugh W (2004) Finding bugs is easy. SIGPLAN Not 39(12):92–106, DOI http://doi.acm.org/10.1145/1052883.1052895

Jungo D, Buchmann D, Nitsche UU (2006) Testing of semantic properties in XML documents. In: Proceedings of the 4th International Workshop on Modelling, Simulation, Verification and Validation of Enterprise Information Systems, Paphos, Cyprus

Kindler E (2004) On the Semantics of EPCs: A Framework for Resolving the Vicious Circle. In: Business Process Management, pp 82–97

Langner P, Schneider C, Wehler J (1997) Relating event-driven process chains to Boolean Petri nets. Report (9707)

Laue R, Awad A (2009) Visualization of business process modeling anti patterns. In: Proceedings of the First International Workshop on Visual Formalisms for Patterns, Electronic Communications of the EASST

Laue R, Kühne S, Gadatsch A (2009) Evaluating the Effect of Feedback on Syntactic Errors for Novice Modellers. In: EPK 2009, Geschäftsprozessmanagement mit Ereignisgesteuerten Prozessketten, CEUR Workshop Proceedings

Layman LM, Williams LA, Amant RS (2008) MimEc: intelligent user notification of faults in the Eclipse IDE. In: CHASE '08: Proceedings of the 2008 international workshop on Cooperative and human aspects of software engineering, ACM, New York, NY, USA, pp 73–76, DOI http://doi.acm.org/10.1145/1370114.1370133

Lindland OI, Sindre G, Sølvberg A (1994) Understanding Quality in Conceptual Modeling. IEEE Softw 11(2):42–49, DOI http://dx.doi.org/10.1109/52.268955

Mendling J (2007) Detection and prediction of errors in EPC business process models. PhD thesis, Vienna University of Economics and Business Administration

Mendling J, Nüttgens M (2003) EPC syntax validation with XML schema languages. In: EPK, pp 19–30

Mendling J, Reijers HA, van der Aalst WMP (2008) Seven process modeling guidelines (7PMG). Tech. Rep. QUT ePrints, Report 12340, Queensland University of Technology

Morimoto S (2008) A survey of formal verification for business process modeling. In: Bubak M, van Albada GD, Dongarra J, Sloot PMA (eds) ICCS (2), Springer, Lecture Notes in Computer Science, vol 5102, pp 514–522

Nüttgens M, Rump FJ (2002) Syntax und Semantik Ereignisgesteuerter Prozessketten (EPK). In: Promise 2002 — Prozessorientierte Methoden

und Werkzeuge für die Entwicklung von Informationssystemen, pp 64–77

Opočenská K, Kopecký M (2008) Incox — a language for XML integrity constraints description. In: DATESO 2008, pp 1–12

Puhlmann F (2007) Soundness verification of business processes specified in the pi-calculus. In: Meersman R, Tari Z (eds) OTM Conferences (1), Springer, Lecture Notes in Computer Science, vol 4803, pp 6–23

Robbins JE, Redmiles DF (2000) Cognitive support, UML adherence, and XMI interchange in Argo/UML. Information & Software Technology 42(2):79–89

Rump FJ (1999) Geschäftsprozeßmanagement auf der Basis ereignisgesteuerter Prozeßketten. B. G. Teubner Verlag Stuttgart Leipzig

Sadiq W, Orlowska ME (2000) Analyzing process models using graph reduction techniques. Information Systems 25(2):117–134

van der Aalst WM (1998) The Application of Petri Nets to Workflow Management. The Journal of Circuits, Systems and Computers 8(1):21–66

van der Aalst WM (1999) Formalization and verification of event-driven process chains. Information & Software Technology 41(10):639–650

van Dongen BF, van der Aalst WM, Verbeek HMW (2005) Verification of EPCs: Using reduction rules and Petri nets. In: CAiSE, pp 372–386

van Dongen BF, Jansen-Vullers M, Verbeek HMW, van der Aalst WM (2007) Verification of the sap reference models using epc reduction, state-space analysis, and invariants. Comput Ind 58(6):578–601, DOI http://dx.doi.org/10.1016/j.compind.2007.01.001

Vanhatalo J, Völzer H, Leymann F (2007) Faster and More Focused Control-Flow Analysis for Business Process Models Through SESE Decomposition. Service-Oriented Computing — ICSOC 2007 pp 43–55, DOI 10.1007/978-3-540-74974-5_4

Vanhatalo J, Völzer H, Koehler J (2008) The refined process structure tree. In: Dumas M, Reichert M, Shan MC (eds) BPM, Springer, Lecture Notes in Computer Science, vol 5240, pp 100–115, URL http://dblp.uni-trier.de/db/conf/bpm/bpm2008.html#VanhataloVK08

Wynn MT (2006) Semantics, verification, and implementation of workflows with cancellation regions and or-joins. PhD thesis, Queensland University of Technology, Brisbane

Wynn MT, Verbeek H, van der Aalst WM, Edmond D (2009) Business process verification — finally a reality! Business Process Management Journal 15(1):74–92

Volker Gruhn, Ralf Laue
Chair of Applied Telematics/e-Business
University of Leipzig
Klostergasse 3
04109 Leipzig
Germany
{gruhn | laue}@ebus.informatik.uni-leipzig.de

Stefan Kühne, Heiko Kern
Business Information Systems
University of Leipzig
Johannisgasse 26
04103 Leipzig
Germany
{kuehne | kern}@informatik.uni-leipzig.de