Patrick Delfmann, Sebastian Herwig, Łukasz Lis, Armin Stein, Katrin Tent, Jörg Becker

Pattern Specification and Matching in Conceptual Models

A Generic Approach Based on Set Operations

Searching for patterns in conceptual models is useful for a number of purposes, for example revealing syntactical errors, model comparison, and identification of business process improvement potentials. In this contribution, we introduce a formal approach for the specification and matching of structural patterns in conceptual models. Unlike existing approaches, we do not focus on a certain application problem or a specific modelling language. Instead, our approach is generic making it applicable for any pattern matching purpose and most conceptual modelling languages. In order to build sets representing structural model patterns, we define formal operations based on set theory, which can be applied to arbitrary models represented by sets. The basic sets represent the model elements, which in turn originate from the modelling language specification's instances. Besides a conceptual and formal specification of our approach, we present particular application examples and a prototypical modelling tool showing its general applicability.

1 Introduction

The structural analysis of conceptual models has multiple applications. To support modellers in their analyses, applying structural patterns to conceptual models is an established approach. Single conceptual models, for example, are analysed using typical error patterns in order to check for syntactical failures (Mendling 2007). In the domain of Business Process Management (BPM), process model analysis helps identifying process improvement potentials (Vergidis et al. 2008). For example, applying structural model patterns to process models can help revealing changes of data medium during process execution (e.g., printing and retyping a document), redundant execution of process activities or application potentials of software systems. Whenever modelling is conducted in a distributed way, model integration is necessary to obtain a coherent view on the modelling domain. To find corresponding fragments and to evaluate integration opportunities, multiple models – generally of the same modelling language – can be compared with each other applying structural model pattern matching (Gori et al. 2005). Different model structures that typically represent equal real-world issues are identified and specified as structurally different, but semantically equal patterns. Counterparts of these patterns are searched via pattern matching in the models to be compared. If pattern counterparts are found in different models, these are marked as candidates for equivalent model sections. A subsequent comparison of their elements reveals whether or not their contents are equal as well. This way, structural pattern matching provides decision support for model comparison and integration.

Model patterns have already been subject of research in the fields of graph theory, database schema integration, and workflow management, to give some examples. However, our literature review reveals that existing pattern matching approaches are limited to a specific domain or restricted to a single modelling language (cf. Sect. 2).

We argue that the modelling community would benefit from a more generic approach which is not restricted to particular modelling languages or application scenarios.

In this article, we present a set theory-based model pattern matching approach, which is generic and thus not restricted regarding its application domain or modelling language. We base this approach on sets and set operations as any model can be regarded as a set of objects and relationships – regardless of the modelling language or application domain. Set operations are used to construct any structural model pattern for any purpose. Therefore, we propose a collection of functions acting on sets of model elements and define set operators to combine the resulting sets of the functions (cf. Sect. 3). This way, we are able to specify structural model patterns for a given modelling language in form of expressions built of the proposed functions and operators. These pattern descriptions can be matched against conceptual models of this language resulting in sets of model elements, which represent particular pattern occurrences. As a specification basis, we use a generic meta-meta model derived from the Meta Object Facility (MOF) specification (OMG 2002, 2006), so we are able to instantiate most common modelling languages. To provide a convenient basis for the specification of patterns, we condense the MOF specification to a generic core. Furthermore, we add an instance section to the type-based MOF specification to allow for the analysis of particular model elements. In this paper, we provide application examples for Event-driven Process Chains (EPC, Scheer 2000) and the Entity-Relationship Model (ERM, Chen 1976; cf. Sect. 4). Furthermore, we present a prototypical modelling tool implementation that shows the applicability of the approach. After a discussion of the benefits and limitations of our approach in Sect. 5, we provide an outlook towards future research in Sect. 6.

2 Related Work

Supporting the structural analysis of conceptual models, fundamental work has been done in the field of graph theory addressing the problem of graph pattern matching (Fu 1995; Gori et al. 2005; Valiente and Martínez 1997; Varró et al. 2006). Based on a given graph, these approaches discuss the identification of structurally equivalent (homomorphism) or synonymous (isomorphism) parts of the given graph in other graphs. Several pattern matching algorithms are proposed, that compute walks through the graphs in order to analyse its nodes and its structure (Dijkman et al. 2009). As a result, they identify patterns representing corresponding parts of the compared graphs (Dijkman 2008). Thus, a pattern is based on a particular labelled graph section. Some approaches are limited to specific types of graphs (e.g., the approaches of Fu (1995) and Varró et al. (2006) are restricted to labelled directed graphs).

In the context of process models, so-called behavioural approaches have been proposed (Hidders et al. 2005; Hirschfeld 1993; de Medeiros et al. 2008). Two process models are considered equivalent if they behave identically during simulation. This implies that the respective modelling languages possess formal execution semantics. Therefore, these approaches are limited to Petri Nets and similar workflow modelling languages (van Dongen et al. 2008). Moreover, due to the requirement of model simulation, these approaches generally consider process models as a whole. Patterns as model subsets are only comparable if they are also executable.

In the domain of database engineering, various approaches have been presented, which address the problem of schema matching. Two input schemas (i.e., descriptions of database structures) are taken and mappings between semantically corresponding elements are established (Rahm and Bernstein 2001). These approaches operate on single elements (Li and Clifton 2000) or assume that the schemas have a tree-like structure (Madhavan et al. 2001). Recently, the methods developed in the context of database schema matching have been applied in the field of ontology

matching as well (Aumueller et al. 2005). Additionally, approaches explicitly dedicated to ontology matching have been presented (Euzenat and Shvaiko 2007). They usually utilise additional context information such as a corresponding collection of documents (Stumme and Mädche 2001). Moreover, as schema-matching approaches operate on approximation-basis, similar structures – and not exact pattern occurrences – are addressed. Consequently, these approaches lack the opportunity of including explicit structure descriptions (e.g., paths of a given length or loops not containing given elements) in the patterns. Design patterns are used in systems analysis and design to describe best-practice solutions for common recurring problems. Common design situations are identified, which can be modelled in various ways. The most desirable solution is identified as a pattern and recommended for further usage. The general idea originates from Alexander et al. (1977), who identified and described patterns in the field of architecture. Gamma et al. (1995) and Fowler (2002) popularised this idea in the domain of object-oriented systems design. Workflow patterns, that is patterns applied to workflow models, is another research domain regarding patterns (van der Aalst et al. 2003). The modeller is expected to adopt the patterns as best-practice and to apply them intuitively whenever a common problem situation is met.

Patterns are also proposed as an indicator for possible conflicts typically occurring in the modelling and model integration process. Hars (1994) proposes a set of general patterns for ERMs. On the one hand, these patterns depict possible structural errors that may occur. For such error patterns the author proposes corresponding patterns which provide correct structures. On the other hand, he discusses sets of model patterns, which possibly lead to conflicts while integrating such models into a total model. Similar work in the field of process modelling has been done by Mendling (2007). Based on the analysis of EPC models, he detects a set of general patterns, which depict common syntactical errors in EPCs. These two approaches focus on particular structural patterns for specific modelling languages.

In contrast to existing approaches, we aim at providing a pattern matching approach that is

- generic to make it applicable to most common modelling languages
- not restricted to particular matching problems
- not restricted to explicit graph sections but also includes recursive structures (e.g., paths of arbitrary length)

3 Specification of Structural Model Patterns

3.1 Sets as a Basis for Pattern Matching

The idea of our approach is to regard a conceptual model as a set of model elements. Here, we further distinguish between objects representing nodes and relationships representing edges interrelating objects. Starting from this set, we search for pattern matches by performing set operations on this basic set. By combining different set operations, patterns are built up successively. Given a pattern definition, the matching process returns a set of model subsets representing the pattern matches found. Every match found is put into a seperate subset. The following example illustrates the general idea.

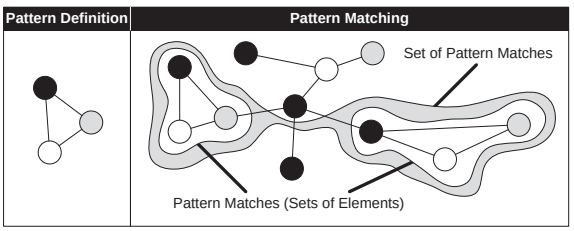

Figure 1: Representation of Pattern Matches through Sets of Elements

In the example, the pattern consists of three objects of different types that are interrelated with each other by relationships (cf. Fig. 1). A pattern match within a model is represented as a set containing three different objects and three relationships that connect them. To distinguish multiple

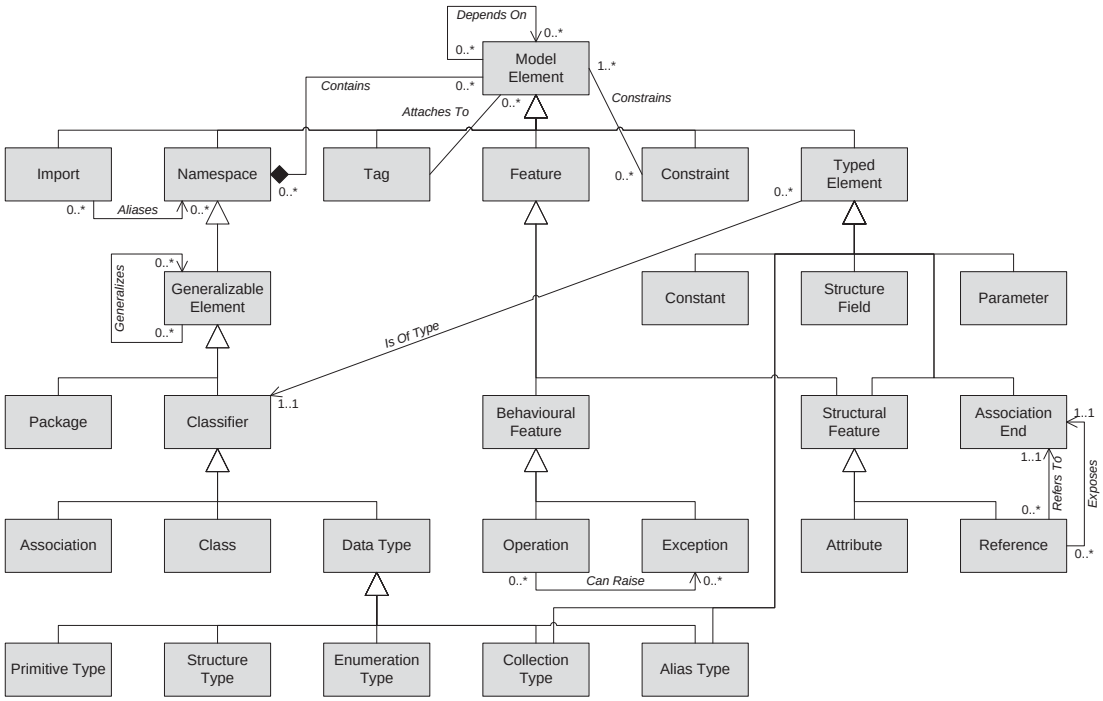

Figure 2: MOF Specification

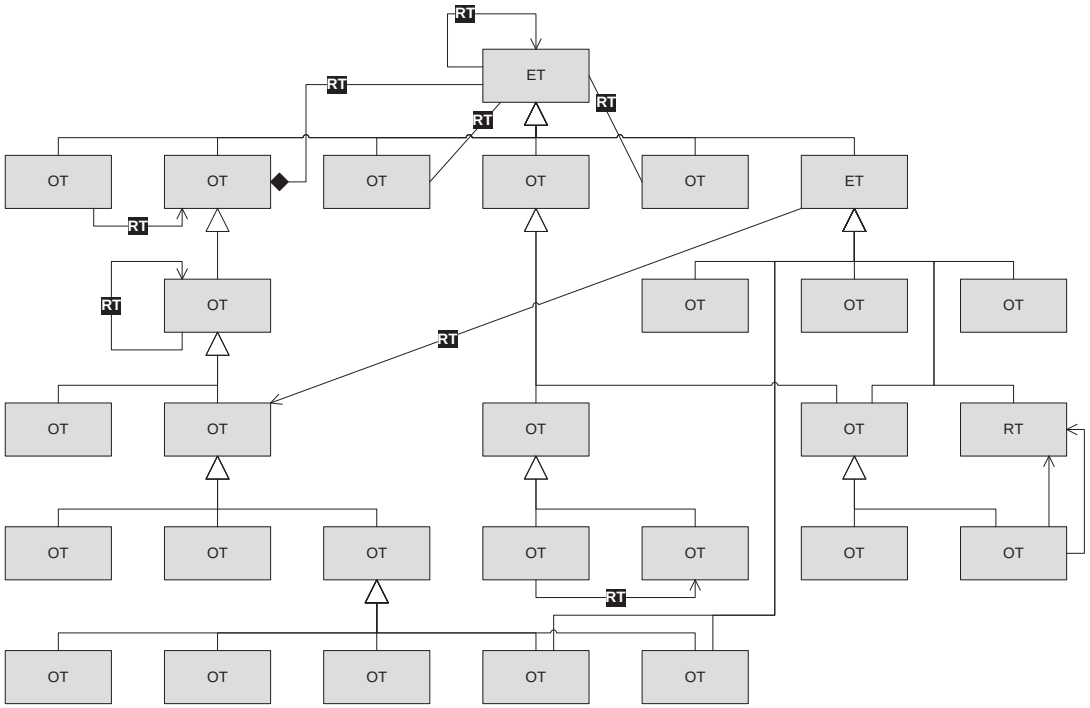

Figure 3: Condensing MOF to a Generic Specification Environment

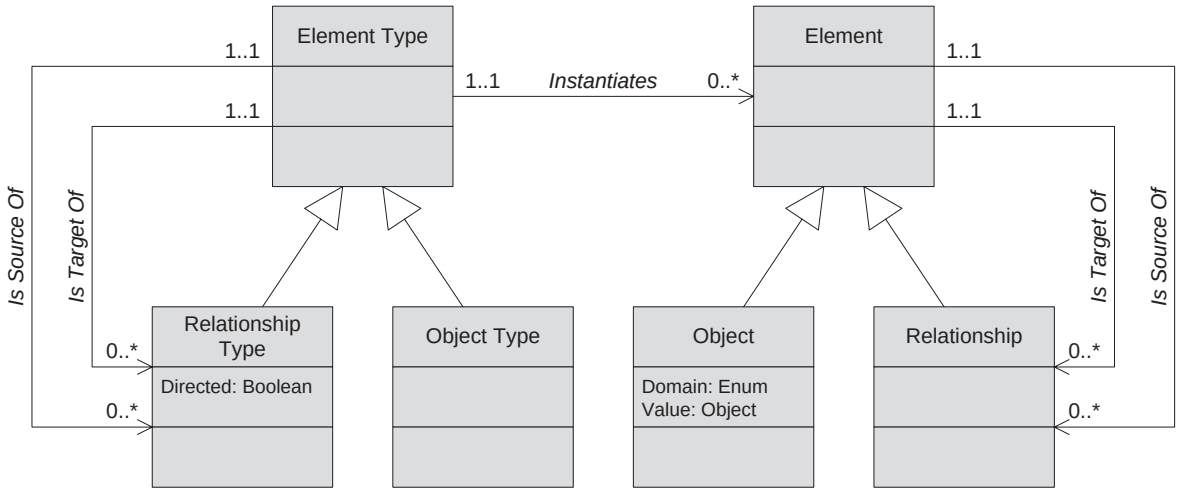

Figure 4: Generic Specification Environment for Conceptual Modelling Languages and Models

pattern matches, each match is represented as a separate subset. Thus, a pattern matching process returns a set of pattern matches (i.e., a set of sets).

3.2 Definition of Basic Sets

As a basis for the definition of structural model patterns, we use a generic specification environment for conceptual modelling languages and models. A popular and established specification environment is provided by the Meta Object Facility (MOF). Within the MOF specification, the central construct is the model element, which is a generic placeholder for any possible construct of a modelling language. This means that it does not only describe the object types (e.g., classes) of a modelling language, but also any other possible construct like relationship types, attributes, or constraints. Every model element can be related to other ones. This way, the MOF specification allows for defining modelling languages with arbitrary expressive power (cf. Fig. 2; for a detailed explanation of MOF constructs, cf. OMG (2006)). In this paper, we use the MOF 1.4 specification. MOF 2.0 consists of 122 classes compared to 29 classes in MOF 1.4 (Bichler 2004). This is why MOF 1.4 provides a more comprehensible overview.

Following the philosophy, that, in general, 'everything can be related to everything', we condensed the MOF specification to a generic core for two specific purposes: First, our pattern matching approach should be generic, thus applicable for any modelling language. Consequently, we aim at being able to search for any construct in models. Hence, we do not need to distinguish these constructs. Second, we aim at providing an easy-to-use and convenient set of constructs for the specification of patterns. Therefore, the number of different constructs should be reduced to a minimum. A similar philosophy is followed for example by the *Graph eXchange Language* (GXL; Holt et al. (2006)), which specifies an XML based exchange format for graphs.

In particular, in order to condense the specification, we rely on graph theory and recognise any conceptual model as a graph G consisting of vertices V and edges E, where $G = (V, E)$ with $E \subseteq V \times V$. Therefore, we reduce the MOF specification to object types representing graph vertices and relationship types representing graph edges. Object types (OT) comprise any of the constructs represented as classes within the MOF specification (with one single exception, see cf. Fig. 3). Relationship types (RT) represent all association edges in the MOF specification as well as the *association end* class, which was introduced by MOF

in order to represent n-ary relationships. Finally, we generalise object types and relationship types as element types (ET). The latter represents the central MOF construct *model element* (cf. Fig. 3).

Our condensed specification consists mainly of these three constructs (cf. Fig. 4). *Element types* are specialised as *object types* (i.e., vertices) and *relationship types* (e.g., edges and links). In some modelling languages, relationships can be interrelated in turn (e.g., association classes in UML Class Diagrams OMG (2009a,b)). To allow these relationships between relationships, we define the relationship type as a specialisation of the element type. Each relationship type has a *source* element type from which it originates, and a *target* element type to which it leads. Relationship types are either *directed* or *undirected*. Whenever the attribute *directed* is *FALSE*, the direction of the relationship type is ignored. As already mentioned above, n-ary relationships have to be represented as well. In most modelling languages providing n-ary relationships, these are represented as vertices (e.g., as a diamond in Class Diagrams and ERMs). Consequently, an n-ary relationship is built by defining an according object type being related to n relationship types and complying with the MOF specification.

The primary purpose of MOF and our condensed specification is to define modelling languages being represented by instances of the specification. As the aim of this paper is to introduce a pattern matching approach for conceptual models (i.e., instances of modelling languages), we need to add a structurally equivalent specification of the models resulting from modelling languages. Therefore, to instantiate element types, object types, and relationship types, we add a specification for *elements*, *objects* and *relationships*. *Elements* are instantiated from their distinct element type. They are specialised as *objects* and *relationships*. Each of the latter leads from a *source element* to a *target element*. Objects can (but do not need to) have *values* which are part of a distinct *domain*. For example, the value of an object 'name' contains the string of the name (e.g., 'product'). As a consequence, the domain of the object 'name' has to be 'string' in this case. Thus, attributes are considered as objects complying with MOF.

For the specification of structural model patterns we define the following sets, elements, and properties originating from the specification environment:

- E: set of all elements available; $e \in E$ is a particular element.
- $\mathcal{P}(E)$: power set of E.
- O: set of all objects available; $O \subseteq E$; $o \in O$ is a particular object.
- R: set of all relationships available; $R \subseteq E$; $r \in R$ is a particular relationship.
- A: set of all element types available; $a \in A$ is a particular element type.
- B: set of all object types available; $B \subseteq A$; $b \in B$ is a particular object type.
- C: set of all relationship types available; $C \subseteq A$; $c \in C$ is a particular relationship type.
- I: set of all instantiations available; $I \subseteq A \times E$; $(a, e) \in I$ is a particular instantiation.
- T: set of all relationship targets available; $T \subseteq E \times R$; $(e, r) \in T$ is a particular target.
- S: set of all relationship sources available; $S \subseteq E \times R$; $(e, r) \in S$ is a particular source.
- X: set of elements with $x \in X \subseteq E$.
- X_k: sets of elements with $X_k \subseteq E$ and $k \in \mathcal{N}_0$.
- x_l: distinct elements with $x_l \in E$ and $l \in \mathcal{N}_0$.
- Y: set of objects with $y \in Y \subseteq O$.
- Z: set of relationships with $z \in Z \subseteq R$.
- *directed(c)*: property *directed* of a particular relationship type c.
- *domain(o)*: property *domain* of a particular object o.
- *value(o)*: property *value* of a particular object o.
- n_x: positive natural number $n_x \in \mathcal{N}_1$.
- R_d: set of all directed relationships available; $R_d \subseteq R$, $((c_d, r_d) \in I \wedge directed(c_d) = TRUE \wedge c_d \in C) \forall r_d \in R_d$

Table 1: Basic Operations

1.	$ElementsOfType(X, a) \subseteq E$ is provided with a set of elements X and a distinct element type a. It returns a set containing all elements of X that belong to the given type:
	$ElementsOfType(X, a) = \{x \in X \mid (a, x) \in I\}$
2.	$ObjectsWithValue(Y, valueY) \subseteq O$ takes a set of objects Y and a distinct value $valueY$. It returns a set containing all objects of Y whose values equal the given one:
	$ObjectsWithValue(Y, valueY) = \{y \in Y \mid value(y) = valueY\}$
3.	$ObjectsWithDomain(Y, domainY) \subseteq O$ takes a set of objects Y and a distinct domain $domainY$. It returns a set with all objects of Y whose domains equal the given one:
	$ObjectsWithDomain(Y, domainY) = \{y \in Y \mid domain(y) = domainY\}$

- T_d: set of all directed relationship targets available;
  $T_d \subseteq T, (r_d \in R_d) \forall (e, r_d) \in T$.
- S_d: set of all directed relationship sources available;
  $S_d \subseteq S, (r_d \in R_d) \forall (e, r_d) \in S$.
- T_u and S_u are undirected counterparts;
  $T_u = T \backslash T_d$ and $S_u = S \backslash S_d$.

3.3 Definition of Set-modifying Functions

Building up structural model patterns successively requires performing set operations on these basic sets. In the following, we introduce predefined functions on these sets in order to provide a convenient specification environment for structural model patterns dedicated to conceptual models. Each function has a defined number of input sets and returns a resulting set. For every function, we specify the input and output sets and provide a formal specification. In addition, we provide textual explanations where necessary. First, since a goal of the approach is to specify any structural pattern, we must be able to reveal specific properties of model elements (e.g., type, value, or value domain; see Tab. 1, 1-3).

Second, relations between elements have to be revealed in order to assemble complex pattern structures successively. Functions are required that combine elements and their relationships and elements that are related respectively. For this purpose, we define helping functions that first return single pattern matches. Second, a further function builds a set containing all the single match sets (see Tab. 2, 1-5).

Third, in order to construct model patterns representing recursive structures (e.g., a path of an arbitrary length consisting of alternating elements and relationships) the following functions are defined. For the specification of recursive structures, we make use of mathematical sequences of the form $(x_i) = (x_1, x_2, ..., x_n), x_i \in E$. However, as our functions generally operate on sets, we need a way to transform sequences into sets. Therefore, we define an auxiliary function $Set((x_i))$ taking a sequence as an input and returning the set containing all members of this sequence: $Set((x_i)) = \{x_i \in E \mid x_i \in (x_i)\} \subseteq E$ (see Tab. 3).

In order to provide a convenient specification environment for structural model patterns, we define some additional functions that are derived from those already introduced (see Tab. 4 and 5).

3.4 Definition of Set Operators for Sets of Sets

By nesting the functions introduced above, it is possible to build up structural model patterns successively. The results of each function can be reused adopting them as an input for other functions. In order to combine different results, the basic set operators *union* ($\cup$), *intersection* ($\cap$), and *complement* ($\backslash$) can be generally used. Since it should be possible to combine not only sets of

Table 2: Operations Relating Elements

1.	$ElementsWithRelations(X, Z) \subseteq \mathcal{P}(E)$ is provided with a set of elements X and a set of relationships Z. It returns a set of sets containing all elements of X and all relationships of Z, which are connected. Each occurrence is represented by an inner set:
	For elements $x_1 \in E : EWR(x_1, Z) = \{z \in Z \mid (x_1, z) \in T \vee (x_1, z) \in S\} \cup \{x_1\}$
	For elements $x \in X : ElementsWithRelations(X, Z) = \{EWR(x, Z)\}$
2.	$ElementsWithOutRelations(X, Z_d) \subseteq \mathcal{P}(E)$ is provided with a set of elements X and a set of relationships Z. It returns a set of sets containing all elements of X that are connected to outgoing relationships of Z, including these relationships. Each occurrence is represented by an inner set:
	For elements $x_1 \in E : EWOR(x_1, Z_d) = \{z_d \in Z_d \mid (x_1, z_d) \in S_d\} \cup \{x_1\}$
	For elements $x \in X : ElementsWithOutRelations(X, Z_d) = \{EWOR(x, Z_d)\}$
3.	$ElementsWithInRelations(X, Z) \subseteq \mathcal{P}(E)$ is defined analogously:
	For elements $x_1 \in E : EWIR(x_1, Z) = \{z_d \in Z_d \mid (x_1, z_d) \in T_d\} \cup \{x_1\}$
	For elements $x \in X : ElementsWithInRelations(X, Z_d) = \{EWIR(x, Z_d)\}$
4.	$ElementsDirectlyRelatedInclRelations(X_1, X_2) \subseteq \mathcal{P}(E)$ is provided with two sets of elements X_1 and X_2. It returns a set of sets containing all elements of X_1 and X_2 that are connected directly via relationships of R, including these relationships. The directions of the relationships given by their 'Source' or 'Target' assignment are ignored. Furthermore, the attribute 'directed' of the according relationship types has to be *FALSE*. Each occurrence is represented by an inner set:
	For elements $x_1 \in X_1 : EDRIR(x_1, X_2) =$ $\{x_2 \in X_2, z \in R_u \mid (x_1, z) \in S_u \wedge (x_2, z) \in T_u \vee (x_2, z) \in S_u \wedge (x_1, z) \in T_u\} \cup \{x_1\}$
	For elements $x_1 \in X_1 : ElementsDirectlyRelatedInclRelations(X_1, X_2) = \{EDRIR(x_1, X_2)\}$
5.	$DirectSuccessorsInclRelations(X_1, X_2) \subseteq \mathcal{P}(E)$ is provided with two sets of elements X_1 and X_2. It returns a set of sets containing all elements of X_1 and X_2 that are connected directly via relationships of R, including these relationships. The directions of the relationships are predefined, that is only relationships from elements of X_1 to elements of X_2 are considered. Each occurrence is represented by an inner set:
	For elements $x_1 \in X_1 : DSIR(x_1, X_2) = \{x_2 \in X_2, z \in R_d \mid (x_2, z) \in S_d \wedge (x_1, z) \in T_d\} \cup \{x_1\}$
	$DirectSuccessorsInclRelations(X_1, X_2) = \{DSIR(x_1, X_2)\}$

pattern matches (i.e., sets of sets) but also the pattern matches themselves, that is, the inner sets, we define additional set operators. These operate on the inner sets of two sets of sets respectively (see Tab. 6).

The *Join* operator performs a *Union* operation on each inner set of the first set with each inner set of the second set. Since we regard patterns as cohesive, only inner sets that have at least one element in common are considered. The *Inner-Intersection* operator *intersects* each inner set of the first set with each inner set of the second set. The *InnerComplement* operator applies a *complement* operation to each inner set of the first outer set combined with each inner set of the second outer set. Only inner sets that have at least one element in common are considered.

As most of the functions introduced in Sect.3.3 expect simple sets of elements as inputs, we introduce further operators that turn sets of sets into simple sets. The *Self-Union* operator merges all inner sets of one set of sets into a single set performing a *union* operation on all inner sets. The *SelfIntersection* operator performs an *intersection* operation on all inner sets of a set of sets successively. The result is a set containing elements that each occur in all inner sets of the original outer set.

Table 3: Operations Returning Element Sequences

1. $Paths(X_1, X_n) \subseteq \mathcal{P}(E)$ takes two sets of elements as input and returns a set of sets containing all sequences which lead from any element of X_1 to any element of X_n. The directions of the relationships, which are part of the paths, are ignored. Furthermore, the attribute 'directed' of the according relationship types has to be *FALSE*. The elements being part of the paths do not necessarily have to be elements of X_1 or X_n, but can also be of $E \backslash X_1 \backslash X_n$. Each path found is represented by an inner set:

   $PX(x_1, x_n) = \{Set(x_1, x_2, ..., x_n) \mid x_2, ..., x_{n-1} \in E \wedge ((x_i, x_{i+1}) \in S_u \vee (x_i, x_{i+1}) \in T_u) \forall 1 \leq i < n\}$

   $Paths(X_1, X_n) = \bigcup_{x_i \in X_1, x_n \in X_n} PX(x_1, x_n)$

   Therefore, we first define a function PX returning all paths, each as a set, starting with a single, particular element $x_1 \in X_1$ and lead to a single, particular element $x_n \in X_n$. Every sequence $(x_1, ..., x_n)$ containing elements $x_1, ..., x_n$ being pairwise related, meaning x_i and x_{i+1} have a source or a target relation, is recognised as a path. Every sequence representing a path is transformed into a set. The result of PX is a set containing sets, which contain the paths leading from x_1 to x_n. Finally, the function PX is executed for every combination of every $x_1 \in X_1$ and $x_n \in X_n$. The resulting outer sets are unified, so the result is a set containing sets of all paths found.

2. $DirectedPaths(X_1, X_n) \subseteq \mathcal{P}(E)$ is the directed counterpart of *Paths*. It returns only paths containing directed relationships of the same direction. Each such path found is represented by an inner set:

   $DPX(x_1, x_n) = \{ Set(x_1, x_2, ..., x_n) \mid x_2, ..., x_{n-1} \in E$
   $\wedge(((x_{2i-1}, x_{2i}) \in S_d \wedge (x_{2i+1}, x_{2i}) \in T_d \forall 1 \leq i \leq \lfloor n/2 \rfloor)$ (a)
   $\vee((x_{2i}, x_{2i-1}) \in T_d \wedge (x_{2i}, x_{2i+1}) \in S_d \forall 1 \leq i \leq \lfloor n/2 \rfloor)$ (b)
   $\vee((x_{2i-1}, x_{2i}) \in S_d \wedge (x_{2i+1}, x_{2i}) \in T_d \wedge (x_{n-1}, x_n) \in S_d \forall 1 \leq i \leq \lfloor n/2 \rfloor - 1)$ (c)
   $\vee((x_{2i}, x_{2i-1}) \in T_d \wedge (x_{2i}, x_{2i+1}) \in S_d \wedge (x_n, x_{n-1}) \in T_d \forall 1 \leq i \leq \lfloor n/2 \rfloor - 1))$ (d)
   $\forall 1 \leq i < n\}$

   $DirectedPaths(X_1, X_n) = \bigcup_{x_1 \in X_1, x_n \in X_n} DPX(x_1, x_n)$

   Therefore, we first define a Function DPX returning all directed paths, each as a set, starting with a single, particular element $x_1 \in X_1$ and lead to a single, particular element $x_n \in X_n$. Every sequence $(x_1, ..., x_n)$ containing objects that are related by relationships having the same direction (i.e., leading from x_1 to x_n), is recognised as a directed path. The formal definition of DPX is divided in four sections (a-d). This is necessary as the DPX function takes elements (not only objects) as inputs. Consequently, a path can either start with an object or with a relationship, and a path can either end with an object or with a relationship (cf. Fig. 5). For example, a path starting with an object and ending with an object requires a "source" relationship between its first two elements, followed by alternating "source" and "target" relationships and ending with a "target" relationship between its last two elements (cf. part (a) of the definition; parts (b)-(d) are defined analogously). Every sequence representing a directed path is transformed into a set. The result of DPX is a set containing sets, which contain the directed paths leading from x_1 to x_n. Finally, the function DPX is executed for every combination of every $x_1 \in X_1$ and $x_n \in X_n$. The resulting outer sets are unified, so the result is a set containing sets of all directed paths found.

3. $Loops(X) \subseteq \mathcal{P}(E)$ takes a set of elements as input and returns a set of sets containing all sequences, which lead from any element of X to itself. The direction of relations and path elements are handled analogously to *Paths*. Each loop found is represented by an inner set:

   $Loops(X) = \bigcup_{x \in X} PX(x, x)$

4. $DirectedLoops(X) \subseteq \mathcal{P}(E)$ is defined analogously:

   $DirectedLoops(X) = \bigcup_{x \in X} DPX(x, x)$

Table 4: Extended Operations (Part 1)

1.	$ElementsWithRelationsOfType(X, Z_d, c_d) \subseteq \mathcal{P}(E)$ is provided with a set of elements X, a set of relationships Z_d and a distinct relationship type c_d. It returns a set of sets containing all elements of X and relationships of Z_d of the type c_d, which are connected. Each occurrence is represented by an inner set:
	$ElementsWithRelationsOfType(X, Z_d, c_d) =$ $ElementsWithRelations(X, ElementsOfType(Z_d, c_d))$
2.	$ElementsWithOutRelationsOfType(X, Z_d, c_d) \subseteq \mathcal{P}(E)$ is provided with a set of elements X, a set of relationships Z_d and a distinct relationship type c_d. It returns a set of sets containing all elements of X that are connected to outgoing relationships of Z_d of the type c_d, including these relationships. Each occurrence is represented by an inner set:
	$ElementsWithOutRelationsOfType(X, Z_d, c_d) =$ $ElementsWithOutRelations(X, ElementsOfType(Z_d, c_d))$
3.	$ElementsWithInRelationsOfType(X, Z_d, c_d) \subseteq \mathcal{P}(E)$ is defined analogously to $ElementsWithOutRelationsOfType$:
	$ElementsWithInRelationsOfType(X, Z_d, c_d) =$ $ElementsWithInRelations(X, ElementsOfType(Z_d, c_d))$
4.	$ElementsWithNumberOfRelations(X, n_x) \subseteq \mathcal{P}(E)$ is provided with a set of elements X and a distinct number n_x. It returns a set of sets containing all elements of X, which are connected to the given number of relationships of R, including these relationships. Each occurrence is represented by an inner set:
	$EWNR(x) = \{r \in R \mid (x, r) \in T \vee (x, r) \in S\} \cup \{x\}$
	$ElementsWithNumberOfRelations(X, n_x) = \{EWNR(x) \mid \lvert EWNR(x)\rvert = n_x + 1\}$
5.	$ElementsWithNumberOfOutRelations(X, n_x) \subseteq \mathcal{P}(E)$ and $ElementsWithNumberOfInRelations(X, n_x) \subseteq \mathcal{P}(E)$ are defined analogously:
	$EWNIR(x) = \{r \in R_d \mid (x, r) \in T_d\} \cup \{x\}$
	$ElementsWithNumberOfInRelations(X, n_x)$ $= \{EWNIR(x) \mid \lvert EWNIR(x)\rvert = n_x + 1\}$
	$EWNOR(x) = \{r \in S_d \mid (x, r) \in S_d\} \cup \{x\}$
	$ElementsWithNumberOfOutRelations(X, n_x)$ $= \{EWNOR(x) \mid \lvert EWNOR(x)\rvert = n_x + 1\}$
6.	$ElementsWithNumberOfRelationsOfType(X, c, n_x) \subseteq \mathcal{P}(E)$ is provided with a set of elements X, a distinct relationship type c, and a distinct number n_x. It returns a set of sets containing all elements of X, which are connected to the given number of relationships of R of the type c, including these relationships. Each occurrence is represented by an inner set:
	$EWNRT(x, c) = \{r \in R \mid (c, r) \in I \wedge ((x, r) \in T \vee (x, r) \in S)\} \cup \{x\}$
	$ElementsWithNumberOfRelationsOfType(X, c, n_x)$ $= \{EWNRT(x, c) \mid \lvert EWNRT(x, c)\rvert = n_x + 1\}$
7.	$ElementsWithNumberOfOutRelationsOfType(X, c_d, n_x)$ and $ElementsWithNumberOfInRelationsOfType(X, c_d, n_x)$, both with codomain $\mathcal{P}(E)$, are defined analogously:
	$EWNIRT(x, c_d) = \{r \in R_d \mid (c_d, r) \in I \wedge (x, r) \in T_d\} \cup \{x\}$
	$ElementsWithNumberOfInRelationsOfType(X, c_d, n_x)$ $= \{EWNIRT(x, c_d) \mid \lvert EWNIRT(x, c_d)\rvert = n_x + 1\}$
	$EWNORT(x, c_d) = \{r \in R_d \mid (c_d, r) \in I \wedge (x, r) \in S_d\} \cup \{x\}$
	$ElementsWithNumberOfOutRelationsOfType(X, c_d, n_x)$ $= \{EWNORT(x, c_d) \mid \lvert EWNORT(x, c_d)\rvert = n_x + 1\}$

Table 5: Extended Operations (Part 2)

8.	$PathsContainingElements(X_1, X_n, X_c) \subseteq \mathcal{P}(E)$ is provided with three sets of elements X_1, X_n, and X_c. It returns a set of sets containing elements that represent all paths from elements of X_1 to elements of X_n, which each contain at least one element of X_c. The direction of relations and path elements are handled analogously to *Paths*. Each path found is represented by an inner set:
	$PCE(x_1, x_n, X_c) = \{Set(x_1, x_2, ..., x_n) \mid x_2, ..., x_{n-1} \in E$
	$\wedge \exists x_c \in \{x_2, ..., x_{n-1}\} \wedge ((x_i, x_{i+1}) \in S_u \vee (x_i, x_{i+1}) \in T_u) \forall 1 \leq i < n\}$
	$PathsContainingElements(X_1, X_n, X_c) = \bigcup_{x_1 \in X_1, x_n \in X_n} PCE(x_1, x_n, X_c)$
	The definition of *PCE* is similar to that of *PX*. In addition, it forces the sequences returned to contain at least one element $x_c \in X_c$ (cf. $\exists x_c \in \{x_2, ..., x_{n-1}\}$ part of the definition).
9.	$DirectedPathsContainingElements(X_1, X_n, X_c) \subseteq \mathcal{P}(E)$, $PathsNotContainingElements(X_1, X_n, X_c) \subseteq \mathcal{P}(E)$, and $DirectedPathsNotContainingElements(X_1, X_n, X_c) \subseteq \mathcal{P}(E)$ are defined analogously:
	$DPCE(x_1, x_n, X_c) = Set((x_1, x_2, ..., x_n)) \mid x_2, ..., x_{n-1} \in E \wedge \exists x_c \in \{x_2, ..., x_{n-1}\} \wedge$
	$(((x_{2i-1}, x_{2i}) \in S_d \wedge (x_{2i+1}, x_{2i}) \in T_d \forall 1 \leq i \leq \lfloor n/2 \rfloor)$
	$\vee((x_{2i}, x_{2i-1}) \in T_d \wedge (x_{2i}, x_{2i+1}) \in S_d \forall 1 \leq i \leq \lfloor n/2 \rfloor)$
	$\vee((x_{2i-1}, x_{2i}) \in S_d \wedge (x_{2i+1}, x_{2i}) \in T_d \wedge (x_{n-1}, x_n) \in S_d \forall 1 \leq i \leq \lfloor n/2 \rfloor - 1)$
	$\vee((x_{2i}, x_{2i-1}) \in T_d \wedge (x_{2i}, x_{2i+1}) \in S_d \wedge (x_n, x_{n-1}) \in T_d \forall 1 \leq i \leq \lfloor n/2 \rfloor - 1)) \forall 1 \leq i < n\}$
	$DirectedPathsContainingElements(X_1, X_n, X_c) = \bigcup_{x_1 \in X_1, x_n \in X_n} DPCE(x_1, x_n, X_c)$
	The definition of *DPCE* is similar to that of *DPX*. In addition, it forces the sequences returned to contain at least one element $x_c \in X_c$ (cf. $\exists x_c \in \{x_2, ..., x_{n-1}\}$ part of the definition).
	$PNCE(x_1, x_n, X_c) = \{Set((x_1, x_2, ..., x_n)) \mid x_2, ..., x_{n-1} \in E \backslash X_c$
	$\wedge((x_i, x_{i+1}) \in S_u \vee (x_i, x_{i+1}) \in T_u) \forall 1 \leq i < n\}$
	$PathsNotContainingElements(X_1, X_n, X_c) = \bigcup_{x_1 \in X_1, x_n \in X_n} PNCE(x_1, x_n, X_c)$
	The definition of *PNCE* is similar to that of *PX*. In addition, it forces the sequences returned not to contain any element $x_c \in X_c$ (cf. $x_2, ..., x_{n-1} \in E \mid X_c$ part of the definition).
	$DPNCE(x_1, x_n, X_c) = \{Set((x_1, x_2, ..., x_n)) \mid x_2, ..., x_{n-1} \in E \backslash X_c$
	$\wedge(((x_{2i-1}, x_{2i}) \in S_d \wedge (x_{2i+1}, x_{2i}) \in T_d \forall 1 \leq i \leq \lfloor n/2 \rfloor)$
	$\vee((x_{2i}, x_{2i-1}) \in T_d \wedge (x_{2i}, x_{2i+1}) \in S_d \forall 1 \leq i \leq \lfloor n/2 \rfloor)$
	$\vee((x_{2i-1}, x_{2i}) \in S_d \wedge (x_{2i+1}, x_{2i}) \in T_d \wedge (x_{n-1}, x_n) \in S_d \forall 1 \leq i \leq \lfloor n/2 \rfloor - 1)$
	$\vee((x_{2i}, x_{2i-1}) \in T_d \wedge (x_{2i}, x_{2i+1}) \in S_d \wedge (x_n, x_{n-1}) \in T_d \forall 1 \leq i \leq \lfloor n/2 \rfloor - 1)) \forall 1 \leq i < n\}$
	$DirectedPathsNotContainingElements(X_1, X_n, X_c) = \bigcup_{x_1 \in X_1, x_n \in X_n} DPNCE(x_1, x_n, X_c)$
	The definition of *DPNCE* is similar to that of *DPX*. In addition, it forces the sequences returned not to contain any element $x_c \in X_c$ (cf. $x_2, ..., x_{n-1} \in E \mid X_c$ part of the definition).
10	$LoopsContainingElements(X, X_c) \subseteq \mathcal{P}(E)$, $DirectedLoopsContainingElements(X, X_c) \subseteq \mathcal{P}(E)$, $LoopsNotContainingElements(X, X_c) \subseteq \mathcal{P}(E)$, and $DirectedLoopsNotContainingElements(X, X_c) \subseteq \mathcal{P}(E)$ are defined analogously:
	$LoopsContainingElements(X, X_c) = \bigcup_{x \in X} PCE(x, x, X_c)$
	$DirectedLoopsContainingElements(X, X_c) = \bigcup_{x \in X} DPCE(x, x, X_c)$
	$LoopsNotContainingElements(X, X_c) = \bigcup_{x \in X} PNCE(x, x, X_c)$
	$DirectedLoopsNotContainingElements(X, X_c) = \bigcup_{x \in X} DPNCE(x, x, X_c)$

Table 6: Set Operators for Sets of Sets

Basic Sets	Operator Definition	Operator Symbol
$F, G \subseteq \mathcal{P}(E), f \in F, g \in G$	$Join(F,G) = \{f \cup g \mid \exists e \in E : e \in f \wedge e \in g\}$	$F \cup\!\!\bullet\, G$
$F, G \subseteq \mathcal{P}(E), f \in F, g \in G$	$InnerIntersection(F,G) = \{f \cap g\}$	$F \cap\!\!\bullet\, G$
$F, G \subseteq \mathcal{P}(E), f \in F, g \in G$	$InnerComplement(F,G) = \{f \backslash g \mid \exists e \in E : e \in f \wedge e \in g\}$	$F \backslash\!\!\bullet\, G$
$F \subseteq \mathcal{P}(E), f \in F$	$SelfUnion(F) = \bigcup_{f \in F} f$	$\bigcup\!\!\bullet\, F$
$F \subseteq \mathcal{P}(E), f \in F$	$SelfIntersection(F) = \bigcap_{f \in F} f$	$\bigcap\!\!\bullet\, F$

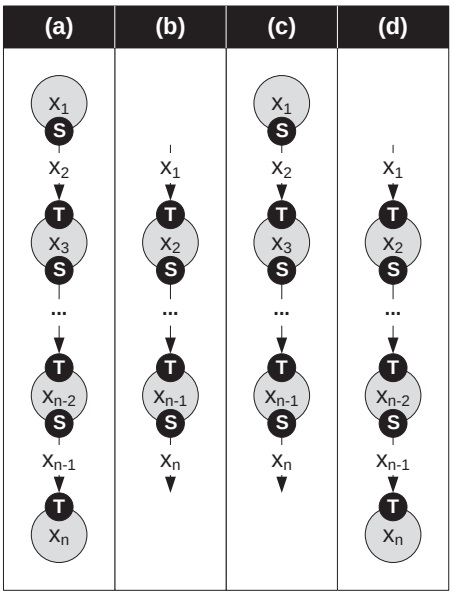

Figure 5: Path Configurations

4 Application

In this section, we demonstrate the generic applicability of our approach based on two exemplary scenarios. Moreover, we present the available tool-support realised by our research prototype.

4.1 Application Examples

To illustrate the usage of the set functions we apply our approach to specify and search for patterns in two exemplary scenarios addressing the modelling languages of EPC and ERM. They can both be specified using MOF, thus they are suitable for an application of the proposed pattern matching approach. Although both EPC and ERM were not *originally* specified using MOF, according specifications have been developed independently (Korherr and List 2007; Lindow et al. 2001) showing the universal applicability of MOF.

EPC Example

In the first example, we regard a simplified modelling language of EPCs. Models of this language consist of the object types *function*, *event*, *AND connector*, *OR connector*, and *XOR connector* (i.e., B = {function, event, AND, OR, XOR}). Furthermore, EPCs consist of different relationship types that lead from any object type to any other object type, except from function to function and from event to event. All these relationship types are directed, (i.e., $directed(c) = TRUE \; \forall \; c \in C$).

A common error in EPCs is that decisions (i.e., XOR or OR splits) are modelled successively to an event. Since events are passive element types of an EPC, they are not able to make a decision (Scheer 2000). Hence, any directed path in an EPC that reaches from an event to a function and contains no further events or functions but an XOR or OR split is a syntax error (cf. Fig. 6).

In order to reveal such errors, we specify an exemplary structural model pattern as depicted in Tab. 7.

The first expression (see Tab. 7, 1) determines all paths that start with an event and end with a function and do not contain any further functions or events. The result is intersected with all paths starting with an event and ending with a function (see Tab. 7, 2) that contain OR and/or XOR connectors (see Tab. 7, 3), but only those that are connected to 2 or more outgoing relationships. Thus, these XORs and ORs are subtracted

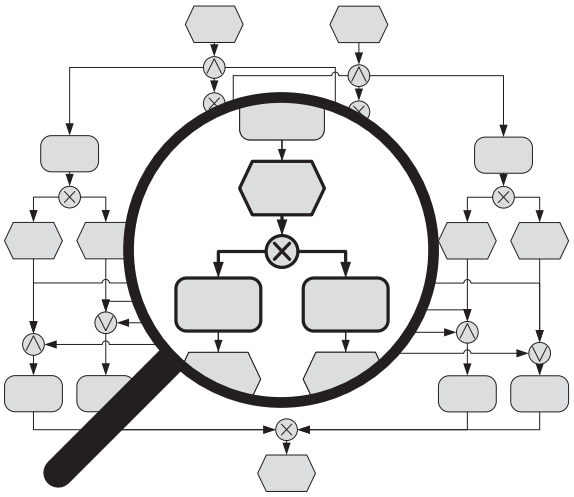

Figure 6: Finding Typical Errors in Event-driven Process Chains

Table 7: Definition of the EPC Model Pattern 'Decision Split After Event'

DirectedPathsNotContainingElements(	1
ElementsOfType(*O*,'Event'),	
ElementsOfType(*O*,'Function'),	
ElementsOfType(*O*,'Event')∪	
ElementsOfType(*O*,'Function')	
)	
∩	
DirectedPathsContainingElements(	2
ElementsOfType(*O*,'Event'),	
ElementsOfType(*O*,'Function'),	
ElementsOfType(*O*,'OR')∪	3
ElementsOfType(*O*,'XOR')	
)	
\	
(*O* ↾ (*ElementsWithNumberOf* ↩	4
OutRelations(	
(*ElementsOfType*(*O*,'XOR')∪	
ElementsOfType(*O*,'OR')),1)	
∪*ElementsWithNumberOfOutRelations*(	
(*ElementsOfType*(*O*,'XOR')∪	
ElementsOfType(*O*,'OR')),0)	
)	
)	

by XORs and ORs that are only connected to one or less relationship(s) (see Tab. 7, 4). Summarising, the matching process returns all paths leading from an event to a function not containing any further events and functions, and containing splitting XOR and/or OR connectors (cf. Fig. 6 and Sect.4.2 for implementation issues and exemplary results).

ERM Example

A second example illustrates the search for typical model structures in data models. For example, in ERMs, so-called *receipt structures* are quite popular. These are commonly used to relate positions of a receipt to its header. Regarding their object types and relationship types, ERMs are defined as follows: B ={EntityType (ET), RelationshipType (RT), RelationalEntityType (RET)} is the set of object types. C ={ET→RT, ET→RET, RET→RET, RET→RT} is the set of relationship types, with $directed(c) = FALSE\ \forall c \in C$. $A = B \cup C$ is the set of element types, and $E = O \cup R$ is the set of particular elements.

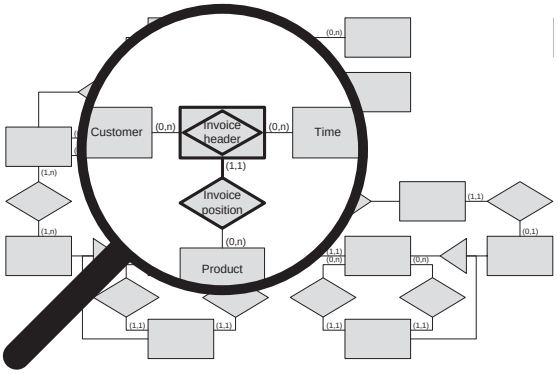

Figure 7: Finding Typical Structures in Entity-Relationship Models

As a task of a schema integration project, a possible subject of analysis could be to find all receipt structures in ERMs containing elements related to the business object 'invoice' (cf. Fig. 7). As a first step, all receipt structures are recognised by the pattern Ψ (cf. Tab. 8, 1 and 2).

The first expression determines all paths beginning with an Entity Type and ending with a Relational Entity Type. These paths are not allowed to contain any further Entity Types, Relational

Table 8: Definition of the ERM Model Pattern 'Receipt Structure'

$\Psi =$

$PathsNotContainingElements($ 1
$ElementsOfType(O,\text{ET}),$
$ElementsOfType(O,\text{RET}),$
$(ElementsOfType(O,\text{ET}) \cup$
$ElementsOfType(O,\text{RET})$
$\cup ElementsOfType(R,\text{ET} \rightarrow \text{RET}))$
$)$
$\circledcirc$
$ElementsDirectlyRelatedInclRelations($ 2
$ElementsWithNumberOfRelations \hookleftarrow$
$OfType($
$ElementsOfType(O,\text{RET}),\text{ET} \rightarrow \text{RET},2)$
$\curvearrowright ElementsOfType(O,\text{RET}),$
$ElementsOfType(O,\text{ET})$
$)$

$\{X \in \Psi \mid$ 3
$ObjectsWithValue(\hookleftarrow$
$ObjectsWithDomain(X,\text{STRING}),$
$\text{"*invoice*"}) \neq \{\}\},$
$\Psi, \Psi' \subseteq \mathcal{P}(E)$

Entity Types, or Edges between Entity Types and Relational Entity Types. Consequently, the first expression returns only paths from an Entity Type across exactly one Relationship Type to a Relational Entity Type. The second expression returns all Relational Entity Types being exactly related to two Entity Types. Therefore, the inner function returns all Relational Entity Types of that nature and the outer one relates them to the Entity Types. Joining both results leads to sets containing receipt structures. As a second step, the resulting pattern sets are restricted to those containing at least one object containing 'invoice' in a string value (cf. Tab. 8, 3).

4.2 Tool Support

In order to show the practical feasibility of our approach, we have implemented a plug-in for a meta modelling tool that was available from a former research project (Delfmann and Knackstedt 2007). The tool consists of a meta modelling environment that is based on the generic specification approach for modelling languages shown in Fig. 4. The plug-in provides a specification environment for structural model patterns. It is integrated into the meta modelling environment of the tool, since the patterns are dependent on the respective modelling language. All basic sets, functions, and set operators introduced in Sect.3 are provided and can be used to build up structural model patterns successively (cf. Fig. 8).

In order to gain a better overview over the patterns, they are displayed and edited in a tree structure. Users can build up the tree-structure through drag-and-drop of the basic sets, functions and set operators. Whenever special characteristics of an according modelling language (function, event etc.) or variables such as numeric values or names are used for the specification, this is expressed by using a 'variable' element (provided by the 'sets' section in the upper left of Fig. 8). The variable element, in turn, is instantiated by selecting a language-specific characteristic from a menu or by entering a particular value (such as '2').

The patterns specified can be applied to any model that is available within the model base and that was developed with the according modelling language. Figure 9 shows an exemplary model that was developed with the modelling language of EPCs and that contains a syntax error consisting of a decision split following an event (cf. Sect. 4.1). The structural model pattern matching process is started by selecting the appropriate pattern to search for. Every match found is displayed by marking the according model section. The user can switch between different matches. In our example, two matches are found, as the decision split following the event leads to two different paths (the second match is shown in the lower right corner of Fig. 9).

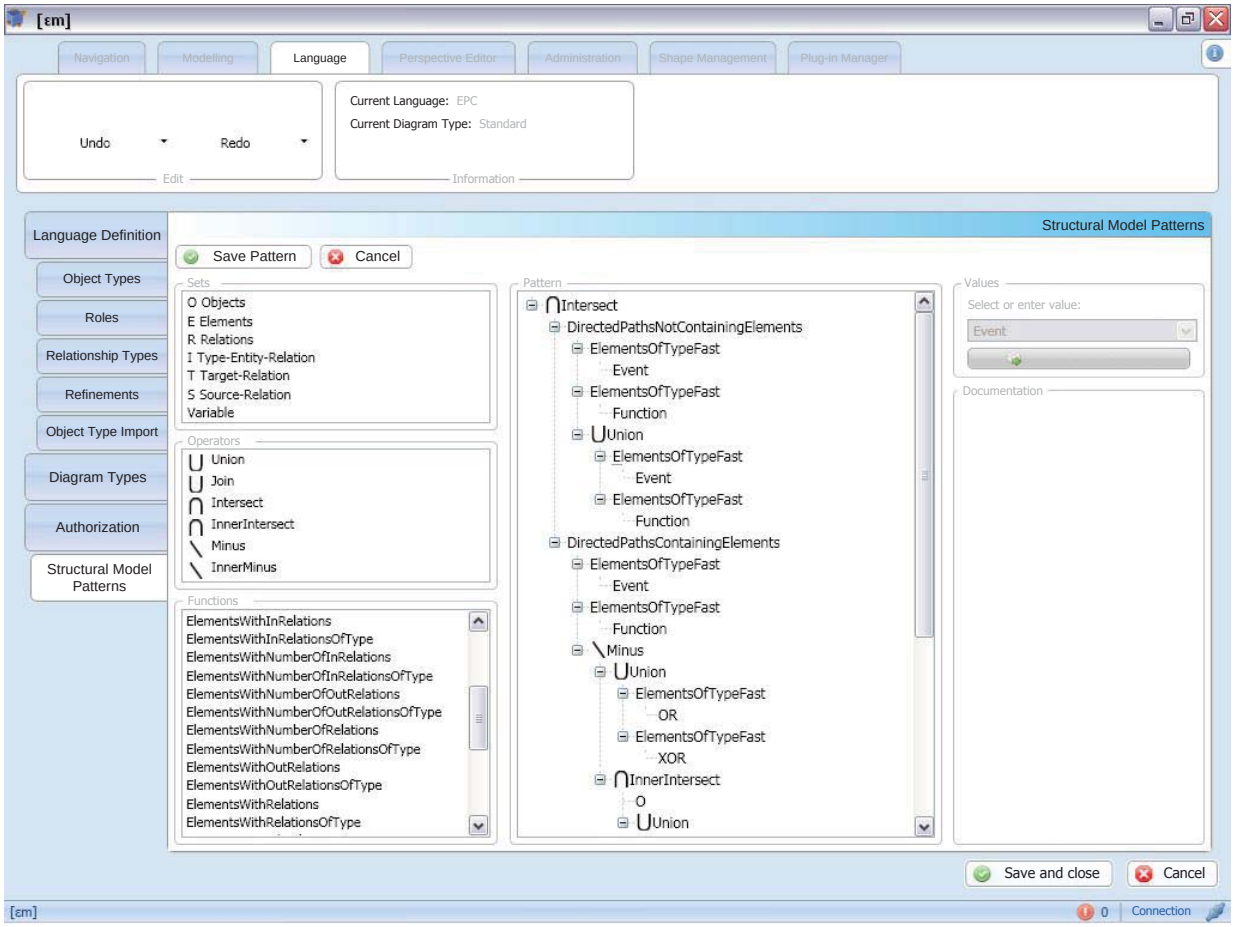

Figure 8: Specification of the Pattern 'Decision Split After Event' to Detect Errors in EPCs

5 Discussion

Supporting model analysis by a generic pattern matching approach is promising, as it is not restricted to a particular problem area or modelling language. Especially when applied in an environment not limited to solely one modelling language (e.g., modelling in an ARIS environment with organisational aspects, data, and processes; cf. Scheer 2000), the flexibility of the approach presented here, in conjunction with the capabilities of a meta-modelling tool, reveals its strengths.

However, the problem of pattern matching in graphs is generally known to be computationally expensive (Bunke 2000), which led to our initial concerns regarding the performance of our approach. In this paper, it is not our goal to calculate the complexity of the algorithm implemented in our approach formally. Instead, to gain an understanding of the user-perceived performance, we conducted a series of performance measurements on different models consisting of 20-150 elements. On a standard dual-core machine with a 1.83 GHz processor and 4 GB RAM and searching for patterns similar to those presented in Sect.4.1, we could measure a net search time of 1-4 milliseconds depending on the model size. Concurrently, retrieving a model from the server and loading it into the client's memory took about 60 times longer and the visualisation of matches took about 40 times longer. We conclude that the search performance was satisfactory from the user's point of view compared to

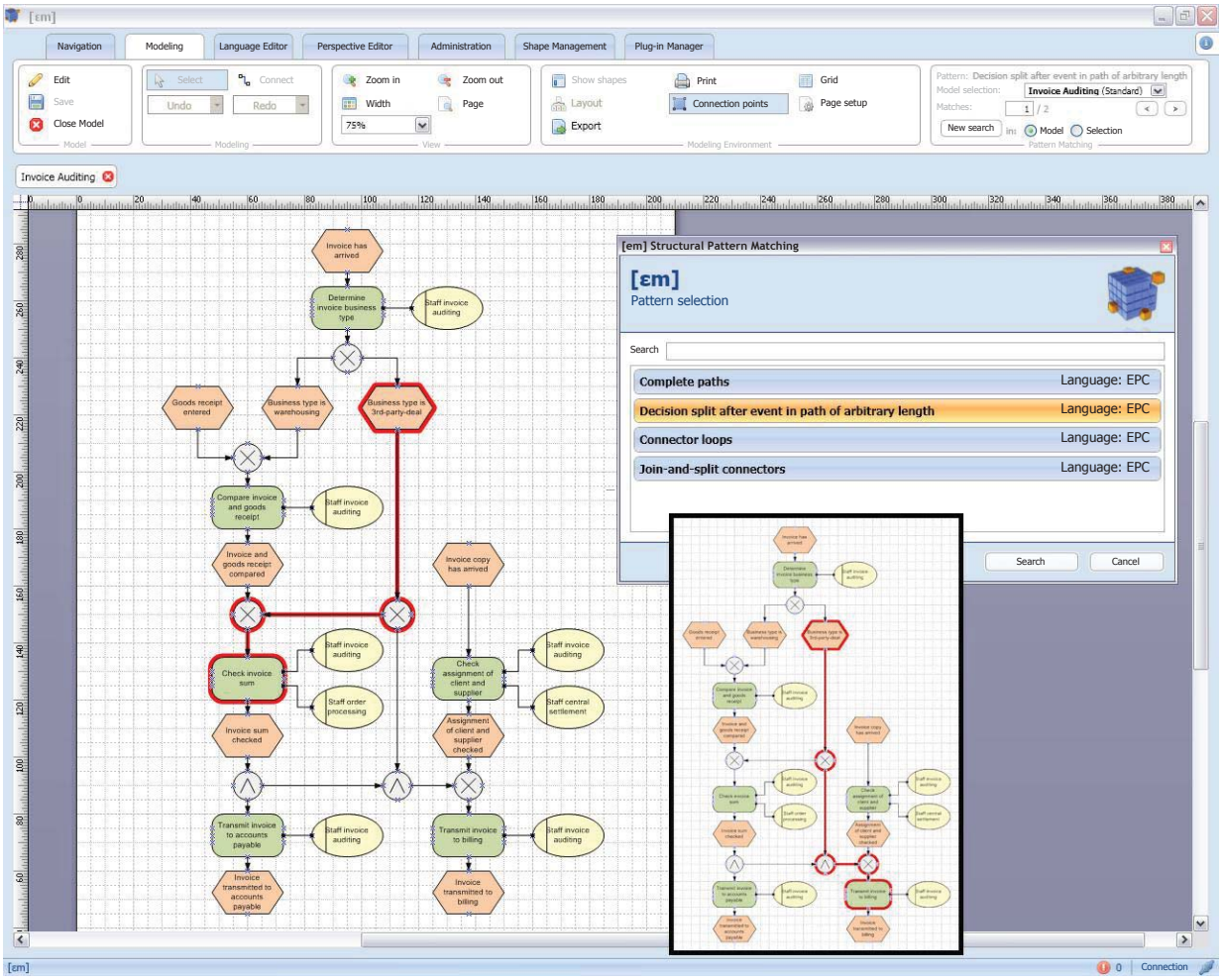

Figure 9: Result of the Pattern Matching Process of 'Decision split after event'

the performance of model loading and matches visualisation.

Although the implementation of the prototype can be regarded as a first rudimentary evaluation step showing the general feasibility of the approach, an in-depth analysis still has to be conducted. This currently happens in a modelling project in the banking sector.

In particular, we are using the approach for revealing weaknesses in business processes in financial institutions. For example, typical shortcomings in business processes are frequent changes in automatic and manual processing, lack of parallelisation and frequent change of organisational responsibility. These weaknesses have been discussed in literature extensively (Baacke et al. 2009; Kusiak et al. 1994). As a starting point, we use these weakness descriptions to define corresponding weakness patterns. The business processes of the institute we are currently investigating are available as process models using different modelling languages (Flow Charts, EPCs and Value Chains). Applying the patterns to the process models and at the same time analysing the process models manually helps us assessing the effectiveness and the efficiency of the approach, compared to a manual analysis.

Since the approach has already shown its general applicability, we expect it to increase the efficiency of weakness analysis of process mod-

els especially when applied to large-scale model bases, compared to a manual analysis. However, up to now, we cannot estimate the actual amount of weaknesses the approach is able to detect in this special case, compared to a manual analysis.

As shown in Sect. 2, a variety of approaches exist, which address the problem of pattern matching in conceptual models. Some of them, implement subgraph isomorphism algorithms (Ullmann 1976) and their later approximate versions based on graph edit distance and maximum common subgraphs (Bunke 2000; Dijkman et al. 2009). In our approach, we do not search for an (exact or approximate) mapping between a graph representing the model and a subgraph representing the pattern. Instead, a pattern is represented by a tree of set functions being an abstract description of a pattern. By using functions such as $Paths(X_1, X_n)$, we are able to define patterns including paths of arbitrary length, which do not have a univocal subgraph representation. We then search for model fragments being exact matches of the pattern, that is, fulfilling the abstract pattern definition.

Therefore, our approach is related to existing EPC syntax checking approaches based on implicit arc types (Mendling and Nüttgens 2003) or PROLOG clauses (Gruhn and Laue 2006). The authors provide abstract descriptions of patterns representing EPC syntax rules (or its violations) using predicate logic and search for exact matches. In our approach, we also conduct exact matching of patterns specified in an abstract form. However, we aim at making our approach applicable to different modelling languages and do not restrict it to syntax checking. As our approach is generic, its performance might be, however, lower than the one of dedicated approaches.

6 Outlook

In the short term, we will focus on completing the evaluation of the presented approach. We will conduct a series of with-without experiments in real-world scenarios. They will show if the presented function set is complete, if the ease of use, especially supported by graphical pattern creation, is satisfactory for users not involved in the development of the approach, and if the application of the approach actually leads to an improved model analysis support.

An obstacle to tackle during future research is the definition of sets. Up to now, the user has to gain deep knowledge on set operations to create the respective patterns. Although the prototype presented above supports the creation of patterns by providing a tree-like structure that can be filled with existing elements like sets, operators and functions (see again Fig. 8), an easier and more convenient way to define patterns could be a graphical one. It is clear to the authors that sets describing patterns like, for example, *circles* or *decision split after event* are complex by nature. However, the initial layout could be generated by "modelling" or "drawing" parts of the pattern, lowering the acceptance barrier for end users. Furthermore, we have revealed conceptual improvement potential during our evaluation project in the banking sector. For example, it could be easier to use explicit model fragments instead of the proposed functions in some situations. Thus, we are going to investigate how we can include explicit model fragments in pattern definitions of our approach.

References

Alexander C., Ishikawa S., Silverstein M. (1977) A Pattern Language. Oxford University Press, New York

Aumueller D., Do H.-H., Massmann S., Rahm E. (2005) Schema and Ontology Matching with COMA++. In: Proceedings of the 2005 ACM SIGMOD international Conference on Management of Data (SIGMOD 2005). New York, pp. 906–908

Baacke L., Becker J., Bergener P., Fitterer R., Greiner U., Stroh F., Räckers M., Rohner P. (2009) Enabling Integration and Optimization of Government Processes With Cross-Functional ICT. In: Weerakkody V., Janssen M., Dwivedi Y. (eds.) Handbook of Research on ICT-Enabled Transformational Government: A Global Perspective. IGI Global, Hershey, New York, pp. 117–139

Bichler L. (2004) Codegeneratoren für MOF-basierte Modellierungssprachen. PhD thesis, Universität der Bundeswehr München, München

Bunke H. (2000) Recent Developments in Graph Matching. In: Proceedings 15th International Conference on Pattern Recognition (ICPR). IEEE Comput. Soc, Barcelona, Spain, pp. 117–124 http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=906030

Chen P. P.-S. (1976) The Entity-Relationship Model – Toward a Unified View of Data. In: ACM Transactions on Database Systems 1(1), pp. 9–36

Delfmann P., Knackstedt R. (2007) Towards Tool Support for Information Model Variant Management – A Design Science Approach. In: Proceedings of the 15th European Conference on Information Systems (ECIS 2007). St. Gallen, pp. 2098–2109

Dijkman R. (2008) Diagnosing Differences between Business Process Models. In: Proceedings of the 6th International Conference on Business Process Management (BPM). Milan, pp. 261–277

Dijkman R., Dumas M., García-Bañuelos L. (2009) Graph Matching Algorithms for Business Process Model Similarity Search. In: Proceedings of the 7th International Conference on Business Process Management (BPM). Ulm

Euzenat J., Shvaiko P. (2007) Ontology Matching. Springer, Berlin

Fowler M. (2002) Patterns of Enterprise Application Architecture. Addison-Wesley, Reading

Fu J. (1995) Pattern Matching in Directed Graphs. In: Galil Z., Ukkonen E. (eds.) Proceedings of the 6th Annual Symposium on Combinatorial Pattern Matching (CPM 1995). Espoo, pp. 64–77

Gamma E., Helm R., Johnson R. E., Vlissides J. (1995) Design Patterns. Elements of Reusable Object-Oriented Software. Addison-Wesley, Amsterdam

Gori M., Maggini M., Sarti L. (2005) The RW2 Algorithm for Exact Graph Matching. In: Singh S., Singh M., Apté C., Perner P. (eds.) Proceedings of the 4th International Conference on Advances in Pattern Recognition (ICAPR 2005). Bath, pp. 81–88

Gruhn V., Laue R. (2006) Validierung syntaktischer und anderer EPK-Eigenschaften mit PROLOG. In: Nüttgens M., Rump F. J., Mendling J. (eds.) 5. Workshop der GI "Geschäftsprozessmanagement mit Ereignisgesteuerten Prozessketten (WI-EPK)". CEUR-WS.org, Viennas, pp. 69–84

Hars A. (1994) Referenzdatenmodelle. Grundlagen effizienter Datenmodellierung. Gabler, Wiesbaden

Hidders J., Dumas M., van der Aalst W. M. P., ter Hofstede A. H. M., Verelst J. (2005) When are two workflows the same? In: Atkinson M., Dehne F. (eds.) Proceedings of the 11th Australasian Symposium on Theory of Computing (CATS 2005). Newcastle, pp. 3–11

Hirschfeld Y. (1993) Petri Nets and the Equivalence Problem. In: Börger E., Y. G., Meinke K. (eds.) Proceedings of the 7th Workshop on Computer Science Logic (CSL 1993). Swansea, pp. 165–174

Holt R. C., Schürr A., Sim S. E., Winter A. (2006) GXL: A graph-based standard exchange format for reengineering. In: Science of Computer Programming 60(2), pp. 149–170

Korherr B., List B. (2007) Extending the EPC and the BPMN with Business Process Goals and Performance Measures. In: Proceedings of the 9th International Conference on Enterprise Information Systems, pp. 287–294

Kusiak A., Larson T. N., Wang J. R. (1994) Reengineering of design and manufacturing processes. In: Computers and Industrial Engi-

neering 26(3), pp. 521–536

Li W.-S., Clifton C. (2000) SemInt: A Tool for Identifying Attribute Correspondences in Heterogeneous Databases using Neural Network. In: Data & Knowledge Engineering 33(1), pp. 49–84

Lindow A., Gogolla M., Richters M. (2001) Ein formal validiertes Metamodell für die Transformation von Schemata in Informationssystemen. In: Bauknecht K., Brauer W., Mück T. (eds.) Proceeding of GI Jahrestagung (GI 2001). Austrian Computer Society, Wien, pp. 662–669

Madhavan J., Bernstein P. A., Rahm E. (2001) Generic schema matching with Cupid. In: Apers P. M. G., Atzeni P., Ceri S., Paraboschi S., Ramamohanarao K., Snodgrass R. T. (eds.) Proceedings of the 27th International Conference on Very Large Data Bases (VLDB 2001). Rome, pp. 49–58

Mendling J. (2007) Detection and Prediction of Errors in EPC Business Process Models. PhD thesis, Wirtschaftsuniversität Wien, Vienna

Mendling J., Nüttgens M. (2003) EPC Modelling based on implicit arc types. In: Godlevsky M, Liddle S. W., Mayr H. C. (eds.) Proceedings of the 2nd International Conference on Information Systems Technology and its Applications (ISTA). Kharkiv, Ukraine

OMG (2002) Meta Object Facility (MOF) Specification. MOF Core specification. Version 1.4. formal/2002-04-03. http://www.omg.org/spec/MOF/1.4/PDF

OMG (2006) Meta Object Facility (MOF) Core Specification. OMG Available Specification. Version 2.0. formal/2006-01-01. http://www.omg.org/spec/MOF/2.0/PDF

OMG (2009a) OMG Unified Modeling Language^{TM} (OMG UML), Infrastructure, Version 2.2. formal/2009-02-04. http://www.omg.org/spec/UML/2.2/Infrastructure/PDF

OMG (2009b) OMG Unified Modeling Language^{TM} (OMG UML), Superstructure, Version 2.2. formal/2009-02-02. http://www.omg.org/spec/UML/2.2/Superstructure/PDF

Rahm E., Bernstein P. A. (2001) A Survey of Approaches to Automatic Schema Matching. In: The International Journal on Very Large Data Bases 10(4), pp. 334–350

Scheer A.-W. (2000) ARIS – Business Process Modelling, 3rd ed. Springer, Berlin

Stumme G., Mädche A. (2001) FCA-Merge: Bottom-up Merging of Ontologies. In: Nebel B. (ed.) Proceedings of the 17th International Joint Conference on Artificial Intelligence (IJCAI 2001). Seattle, pp. 225–230

Ullmann J. R. (1976) An Algorithm for Subgraph Isomorphism. In: Journal of the ACM (JACM) 23(1), pp. 31–42

Valiente G., Martínez C. (1997) An Algorithm for Graph Pattern-Matching. In: Baeza-Yates R., Ziviani N. (eds.) Proceedings of the 4nd South American Workshop on String Processing (WSP 1997). Brighton, pp. 180–197

Varró G., Varró D., Schürr A. (2006) Incremental Graph Pattern Matching – Data Structure and Initial Experiments. In: Margaria T., Padberg J., Taentzer G. (eds.) Proceedings of the 2nd International Workshop on Graph and Model Transformation (GraMoT 2006). Brighton

Vergidis K., Tiwari A., Majeed B. (2008) Business Process Analysis and Optimization: Beyond Reengineering. In: IEEE Transactions on Systems, Man, and Cybernetics 38(1), pp. 69–82

de Medeiros A. K. A., van der Aalst W. M. P., Weijters A. J. M. M. (2008) Quantifying Process Equivalence based on Observed Behavior. In: Data & Knowledge Engineering 64(1), pp. 55–74

van Dongen B., Dijkman R., Mendling J. (2008) Measuring Similarity between Business Process Models. In: Proceedings of the 20th International Conference on Advanced Information Systems Engineering (CAiSE 2008). Montpellier, pp. 450–464

van der Aalst W. M. P., ter Hofstede A. H. M., Kiepuszewski B., Barros A. P. (2003) Workflow Patterns. In: Distributed and Parallel Databases 14(3), pp. 5–51

Patrick Delfmann, Sebastian Herwig, Łukasz Lis, Armin Stein, Jörg Becker

European Research Center for Information Systems (ERCIS)
University of Münster
Leonardo-Campus 3
48149 Münster
Germany
{patrick.delfmann | sebastian.herwig | lukasz.lis | armin.stein | joerg.becker}@ercis.uni-muenster.de

Katrin Tent

Institut für Mathematische Logik und Grundlagenforschung
University of Münster
Einsteinstrasse 62
48149 Münster
Germany
tent@math.uni-muenster.de