A Generic Set Theory-Based Pattern Matching Approach for the Analysis of Conceptual Models

Jörg Becker, Patrick Delfmann, Sebastian Herwig, and Łukasz Lis

University of Münster, European Research Center for Information Systems (ERCIS),
Leonardo-Campus 3, 48149 Münster, Germany
{becker,delfmann,herwig,lis}@ercis.uni-muenster.de

Abstract. Recognizing patterns in conceptual models is useful for a number of purposes, like revealing syntactical errors, model comparison, and identification of business process improvement potentials. In this contribution, we introduce an 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 modeling language. Instead, our approach is generic making it applicable for any pattern matching purpose and any conceptual modeling language. In order to build sets representing structural model patterns, we define operations based on set theory, which can be applied to arbitrary sets of model elements and relationships. Besides a conceptual specification of our approach, we present a prototypical modeling tool that shows its applicability.

Keywords: Conceptual Modeling, Pattern Matching, Set Theory.

1 Introduction

The structural analysis of conceptual models has multiple applications. For example, single conceptual models are analyzed in order to check for syntactical errors [1]. In the domain of Business Process Management (BPM), process model analysis helps identifying process improvement potentials [2]. Whenever modeling is conducted in a distributed way, model integration is necessary to obtain a coherent view on the modeling domain. Multiple models are compared with each other to find corresponding fragments and to evaluate integration opportunities [3].

Structural model patterns can be applied in these scenarios to support modelers in their analyses. In the BPM domain, for example, model patterns can help identifying media disruptions, lack of parallelism, or redundancies. Model patterns have already been subject of research in the fields of database schema integration and workflow management, to give some examples. However, our literature review shows that existing approaches are limited to a specific domain or restricted to a single modeling language (cf. Section 2). We argue that the modeling community would benefit from a more generic approach, which is not limited to particular modeling languages or application scenarios.

In this paper, we present a set theory-based model pattern matching approach, which is generic and thus not restricted regarding its application domain or modeling language. We base this approach on set theory as any model can be regarded as a set of objects

A.H.F. Laender et al. (Eds.): ER 2009, LNCS 5829, pp. 41–54, 2009.

and relationships – regardless of the model's 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. Section 3). This way, we are able to specify structural model patterns in form of expressions built up of the proposed functions and operators. These pattern descriptions can be matched against conceptual models resulting in sets of model elements, which represent particular pattern occurrences. As a specification basis, we use a generic meta-meta model being able to instantiate any modeling language. To illustrate the application of the approach, we provide an application example for Event-driven Process Chains (EPCs) [4] (cf. Section 4) and present a prototypical modeling tool implementation (cf. Section 5). Finally, we conclude our paper and outline the further need for research (cf. Section 6).

2 Related Work

Fundamental work is done in the field of graph theory addressing the problem of graph pattern matching [3, 6, 7]. 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. To identify such parts, several pattern matching algorithms are proposed, which compute walks through the graphs in order to analyze the nodes and the structure of the graphs. As a result, they recognize patterns representing corresponding parts of the compared graphs. Thus, a pattern is based on a particular labeled graph section and is not predefined independently. Some approaches are limited to specific types of graphs (e.g., the approaches of [5, 6] are restricted to labeled directed graphs).

In systems analysis and design, so-called *Design Patterns* are used to describe best-practice solutions for common recurring problems. Common design situations are identified, which can be modeled in various ways. The most desirable solution is identified as a pattern and recommended for further usage. The general idea originates from [7], who identified and described patterns in the field of architecture. [8] and [9] popularized this idea in the domain of object-oriented systems design. Workflow Patterns is another dynamically developing research domain regarding patterns [10]. However, the authors of these approaches do not consider pattern matching. Instead, the modeler is expected to manually adopt the patterns as best-practice and to apply them intuitively whenever a common problem situation is met. An implementable pattern matching support is not addressed.

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 produced [11]. These approaches operate on single elements only [12] or assume that the schemas have a tree-like structure [13]. Recently, the methods developed in the context of database schema matching have been applied in the field of ontology matching as well [14]. Additionally, approaches explicitly dedicated to matching ontologies have been presented. They usually utilize additional context information (e.g., a corresponding a collection of documents [15]), which is not given in standard conceptual modeling settings. Moreover, as schema matching approaches operate on approximation basis, only similar structures – and not exact pattern

occurrences – are addressed [16]. 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.

Patterns are also proposed as an indicator for possible conflicts typically occurring during the modeling and model integration process. [17] proposes a collection of general patterns for Entity-Relationship Models (ERMs [18]). On the one hand, these patterns depict possible structural errors. For such error patterns corresponding patterns are proposed, which provide correct structures. On the other hand, a number of model patterns is discussed, which possibly lead to conflicts while integrating such models into a total model. Similar work in the field of process modeling is done by [1]. Based on the analysis of EPCs, he detects a collection of general patterns, which depict syntactical errors in EPCs.

In the context of process modeling, so-called behavioral approaches have been proposed [19, 20, 21]. Two process models are considered equivalent if they behave identically during simulation. This implies that the respective modeling languages possess formal execution semantics. Therefore, the authors focus on Petri Nets and related languages [22]. 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.

Summarizing, applying the analyzed approaches to pattern matching in conceptual models in the contexts outlined in Section 1 leads to a set of restrictions outlined in Table 1.

Table 1. Restrictions of Existing Pattern Approaches

Category	Restriction, Approach
Special preconditions	• Only directed graphs [5, 6] • Only acyclic models [13] • Additional text mining is required [15] • Only suitable for executable models [19, 20, 21, 22]
Similarity-based matching	• Similarity check rather than exact matching [12, 13, 14, 15] • Only patterns with defined number of elements (no paths of arbitrary length etc.) [12, 13, 14, 15] • No element type match, only particular element match [12, 13, 14, 15]
No matching support	• Patterns for reuse [8, 9, 10] • Syntax error patterns [1, 17]

In contrast, we aim at supporting pattern matching in conceptual models of any modeling language and for any type of patterns (i.e., patterns with a predefined or unlimited number of elements). In particular, these are patterns like “activity precedes activity” as well as “path starts and ends with activity”. Furthermore, a pattern matching process should return every model section (e.g., *activity 1* precedes *activity 2*) representing an exact match of the pattern (e.g., *activity* precedes *activity*).

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 and relationships. Starting from this set, pattern matches are searched by performing

set operations on this basic set. By combining different set operations, the pattern is built up successively. Every match found is put into an own set. The following example demonstrates our approach in general.

A pattern definition consists of three objects of different types that are interrelated with each other by relationships. An according pattern match within a model is represented as a set containing three different objects and three relationships that connect them. To distinguish multiple pattern matches, each match is represented as an own set of elements. Thus, the result of a pattern matching process is represented by a set of pattern matches (i.e., a set of sets, cf. Fig. 1).

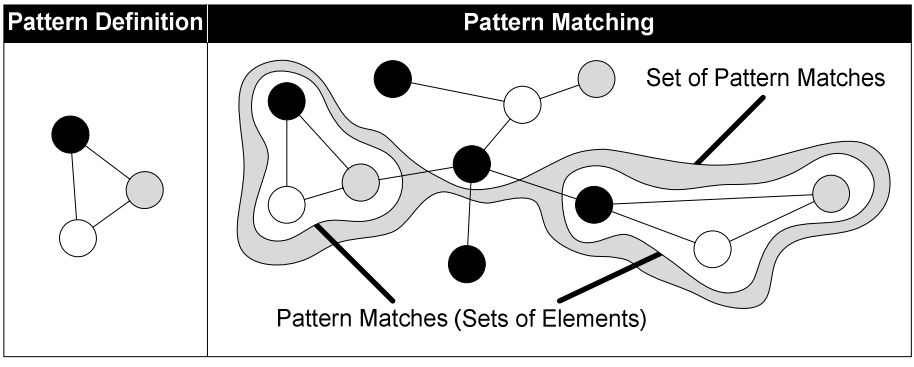

Fig. 1. Representation of Pattern Matches through Sets of Elements

3.2 Definition of Basic Sets

As a basis for the specification of structural model patterns, we use a generic meta-meta model for conceptual modeling languages (cf. Fig. 2), which is closely related to the Meta Object Facility (MOF) specification [23]. Here, we only use a subset, which is represented in the Entity-Relationship notation with (min,max)-cardinalities [24].

Modeling languages typically consist of modeling objects that are interrelated through relationships (e.g., vertices and edges). In some languages, relationships can be interrelated in turn (e.g., association classes in UML Class Diagrams [25]).

Hence, modeling languages consist of *element types*, which are specialized as *object types* (e.g., nodes) and their *relationship types* (e.g., edges and links). In order to allow relationships between relationships, the relationship type is defined as a specialization 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. The instantiation of modeling languages leads to models, which consist of particular *elements*. These are *instantiated* from their distinct element type. Elements are specialized into *objects* and *relationships*. Each of the latter leads from a *source* element to a *target* element. Objects can have *values,* which are part of a distinct *domain.* For example, the value of an object "name" contains the

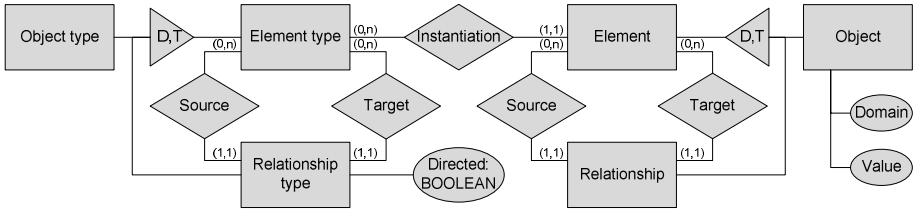

Fig. 2. Generic Specification Environment for Conceptual Modeling Languages and Models

string of the name (e.g., "product"). As a consequence, the domain of the object "name" has to be "string". Thus, attributes are considered as objects.

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

- E: finite set of all elements; $e \in E$ is a particular element.
- O: finite set of all objects with $O \subseteq E$; $o \in O$ is a particular object.
- R: finite set of all relationships with $R \subseteq E$; $r \in R$ is a particular relationship.
- A: finite set of all element types; $a \in A$ is a particular element type.
- B: finite set of all object types with $B \subseteq A$; $b \in B$ is a particular object type.
- C: finite set of all relationship types with $C \subseteq A$; $c \in C$ is a particular relationship type.

In addition, we introduce the following notations, which are needed for the specification of set-modifying functions (cf. Section 3.3):

- X: set of elements with $x \in X \subseteq E$.
- X_k: sets of elements with $X_k \subseteq E$ and $k \in \mathbf{N_0}$
- Y: set of objects with $y \in Y \subseteq O$.
- Z: set of relationships with $z \in Z \subseteq R$.
- n_X: positive natural number $n_X \in \mathbf{N_1}$

3.3 Definition of Set-Modifying Functions

Building up structural model patterns successively requires performing set operations on the basic sets. In the following, we introduce predefined functions on these sets in order to provide a convenient specification environment dedicated to conceptual models. However, in order to make the approach reusable for multiple purposes, the formal specification of these functions is based on *predicate logic*. For clarity reasons, we will not present the detailed formal specifications here. We rather present the functions as black boxes and exclusively focus on their input and output sets.

Each function has a defined number of input sets and returns a resulting set. 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):

- *ElementsOfType(X,a)* is provided with a set of elements *X* and a distinct element type *a*. It returns a set of all elements of *X* that belong to the given element type.
- *ObjectsWithValue(Y,value$_Y$)* is provided with a set of objects *Y* and a distinct value *value$_Y$*. It returns a set of all objects of *Y* whose values equal the given one.
- *ObjectsWithDomain(Y,domain$_Y$)* takes a set of objects *Y* and a distinct domain *domain$_Y$*. It returns a set of all objects of *Y* whose domains equal the given one.

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.

- *ElementsWithRelations(X,Z)* 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 undirected relationships of Z, which are connected. Each occurrence is represented by an inner set.
- *ElementsWithOutRelations(X,Z)* 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 directed, outgoing relationships of Z, including these relationships. Each occurrence is represented by an inner set.
- *ElementsWithInRelations(X,Z)* is defined analogously to *ElementsWithOutRelations*. In contrast, it only returns incoming relationships.
- *ElementsDirectlyRelatedInclRelations(X_1,X_2)* 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.
- *DirectSuccessorsInclRelations(X_1,X_2)* is defined analogously to *ElementsDirectlyRelatedInclRelations.* In contrast, it only returns relationships that are directed, whereas the source elements are part of X_1 and the target elements are part of X_2.

Third, 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:

- *Paths(X_1,X_n)* 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, given by their "Source" or "Target" assignment, are ignored. Furthermore, the attribute "directed" of the according relationship types has to be FALSE. The elements that are 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.
- *DirectedPaths(X_1,X_n)* is defined analogously to *Paths.* In contrast, it only returns directed paths leading from X_1 to X_n.

- *Loops(X)* 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 directions of the relationships, which are part of the loops, given by their "Source" or "Target" assignment, are ignored. Furthermore, the attribute "directed" of the according relationship types has to be FALSE. The elements that are part of the loops do not necessarily have to be elements of *X,* but can also be of *E\X*. Each loop found is represented by an inner set.

- *DirectedLoops(X)* is defined analogously to *Loops.* In contrast, it only returns loops, the relationships of which have the same direction.

To avoid infinite sets, only finite paths and loops are returned. As soon as there exists a complete sub-loop on a loop or a path, and this sub-loop is passed the second time, the search aborts. The path or loop that was searched for is excluded from the result set. To provide a convenient specification environment for structural model patterns, we define some additional functions that are derived from those already introduced:

- *ElementsWithRelationsOfType(X,Z,c)* is provided with a set of elements X, a set of relationships Z and a distinct relationship type c. It returns a set of sets containing all elements of X and relationships of Z of the type c, which are connected. Each occurrence is represented by an inner set.

- *ElementsWithOutRelationsOfType(X,Z,c)* is provided with a set of elements X, a set of relationships Z and a relationship type c. It returns a set of sets containing all elements of X that are connected to outgoing relationships of Z of the type c, including these relationships. Each occurrence is represented by an inner set.

- *ElementsWithInRelationsOfType(X,Z,c)* is defined analogously to *ElementsWithOutRelationsOfType*.

- *ElementsWithNumberOfRelations(X,n_X)* 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.

- *ElementsWithNumberOfOutRelations(X,n_X)* 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 outgoing relationships of *R*, including these relationships. Each occurrence is represented by an inner set.

- *ElementsWithNumberOfInRelations(X,n_X)* is defined analogously to *ElementsWithNumberOfOutRelations.*

- *ElementsWithNumberOfRelationsOfType(X,c,n_X)* 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.

- *ElementsWithNumberOfOutRelationsOfType(X,c,n_X)* 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 outgoing relationships of R of the type c, including these relationships. Each occurrence is represented by an inner set.

- *ElementsWithNumberOfInRelationsOfType(X,c,n_X)* is defined analogously to *ElementsWithNumberOfOutRelationsOfType*.
- *PathsContainingElements(X_1,X_n,X_c)* 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 elements that are part of the paths do not necessarily have to be elements of X_1 or X_n, but can also be of $E\backslash X_1\backslash X_n$. The directions of the relationships, which are part of the paths, given by their "Source" or "Target" assignment, are ignored. Furthermore, the attribute "directed" of the according relationship types has to be FALSE. Each such path found is represented by an inner set.
- *DirectedPathsContainingElements(X_1,X_n,X_c)* is defined analogously to *PathsContainingElements*. In contrast, it only returns directed paths containing at least one element of X_c and leading from X_1 to X_n.
- *PathsNotContainingElements(X_1,X_n,X_c)* is defined analogously to *PathsContainingElements*. It returns only paths that contain no elements of X_c.
- *DirectedPathsNotContainingElements(X_1,X_n,X_c)* is defined analogously to *DirectedPathsContainingElements*. It returns only paths that contain no elements of X_c.
- *LoopsContainingElements(X,X_c)* is defined analogously to *PathsContainingElements*.
- *DirectedLoopsContainingElements(X,X_c)* is defined analogously to *LoopsContainingElements*. In contrast, it only returns directed loops containing at least one element of X_c.
- *LoopsNotContainingElements(X,X_c)* is defined analogously to *LoopsContainingElements*. It returns only those loops that contain no elements of X_c.
- *DirectedLoopsNotContainingElements(X,X_c)* is defined analogously to *DirectedLoopsContainingElements*. It returns only loops that contain no elements of X_c.

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 the functions 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 (\) can be used in general. Since it should be possible to combine not only sets of pattern matches (i.e., sets of sets) but also the pattern matches themselves (i.e., the inner sets), we define additional set operators. These operate on the inner sets of two sets of sets respectively.

The UNION operator combines the elements of a set. Applied to sets of sets, it simply puts the inner sets of two sets into a resulting set (cf. Fig. 3).

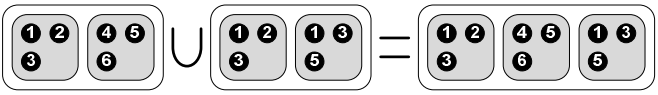

Fig. 3. UNION Operator

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 (cf. Fig. 4).

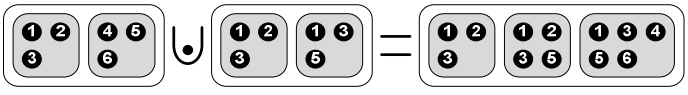

Fig. 4. JOIN Operator

The INTERSECTION operator compares the elements of two sets. Only elements that occur in both sets are put into the resulting set. Applied to sets of sets, it puts the inner sets of two sets containing exactly the same elements into a resulting set (cf. Fig. 5).

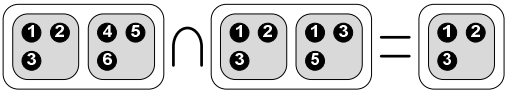

Fig. 5. INTERSECTION Operator

The INNER_INTERSECTION operator INTERSECTs each inner set of the first set with each inner set of the second set (cf. Fig. 6).

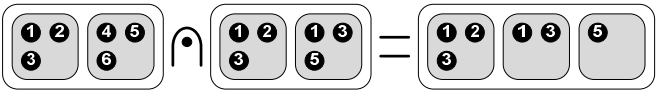

Fig. 6. INNER_INTERSECTION Operator

Applying the COMPLEMENT operator, elements occurring in the first set are removed if they occur in the second set as well. Applied to sets of sets, inner sets of the first outer set are removed, if they occur in the second outer set as well (cf. Fig. 7).

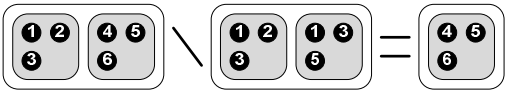

Fig. 7. COMPLEMENT Operator

The INNER_COMPLEMENT operator applies a COMPLEMENT operation to each inner set of the first outer set with each inner set of the second outer set. Only inner sets that have at least one element in common are considered (cf. Fig. 8)

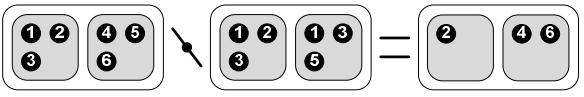

Fig. 8. INNER_COMPLEMENT Operator

Since most of the functions introduced in Section 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 to all inner sets (cf. Fig. 9).

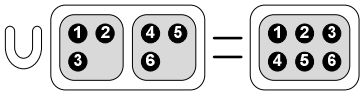

Fig. 9. SELF_UNION Operator

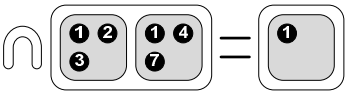

Fig. 10. SELF_INTERSECTION Operator

The SELF_INTERSECTION operator is defined analogously. It performs an INTERSECTION operation to 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 set (cf. Fig. 10).

4 Application of Structural Model Patterns

To illustrate the usage of the set functions, we apply our pattern matching approach to syntax verification in EPCs. Therefore, we regard a simplified modeling 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., *c.directed=TRUE* $\forall c \in C$).

A common error in EPCs is that decision splits are modeled successively to an event. Since events are passive element types of an EPC, they are not able to make a decision [4]. Hence, any directed path in an EPC that reaches from an event to a function and contains no further events or functions but a XOR or OR split is a syntax error. In order to reveal such errors, we specify the following structural model pattern:

DirectedPathsNotContainingElements (*ElementsOfType (O, 'Event'),* *ElementsOfType (O, 'Function'),* *(ElementsOfType (O, 'Event') UNION ElementsOfType (O, 'Function')))*	1
INTERSECTION	
DirectedPathsContainingElements (*ElementsOfType (O, 'Event'),* *ElementsOfType (O, 'Function'),*	2
((ElementsOfType (O, 'OR') UNION ElementsOfType(O, 'XOR'))	3

```
    COMPLEMENT
    ( O INNER_INTERSECTION ( ElementsWithNumberOfOutRelations (
        ( ElementsOfType (O, 'XOR') UNION ElementsOfType (O, 'OR') ), 1)
    UNION ElementsWithNumberOfOutRelations (
        ( ElementsOfType (O, 'XOR') UNION ElementsOfType (O, 'OR') ), 0)
) ) ) )
```

4

The first expression (cf. 1st block) 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* (cf. 2nd block) that contain *OR* and/or *XOR* connectors (cf. 3rd block), but only those that are connected to two or more outgoing relationships. Thus, these *XORs* and *ORs* are subtracted by *XORs* and *ORs* that are only connected to one or less relationship(s) (cf. 4th block). Summarizing, all paths are returned that lead from an *event* to a *function* not containing any further *events* and *functions*, and that contain splitting *XOR* and/or *OR* connectors (cf. Section 5 for implementation issues and exemplary results). This way, any syntax error pattern can be specified and applied to any model base.

5 Tool Support

In order to show the feasibility of the approach, we have implemented a plug-in for a meta modeling tool that was available from a former research project [26]. The tool consists of a meta modeling environment that is based on the generic specification approach for modeling languages shown in Fig. 2.

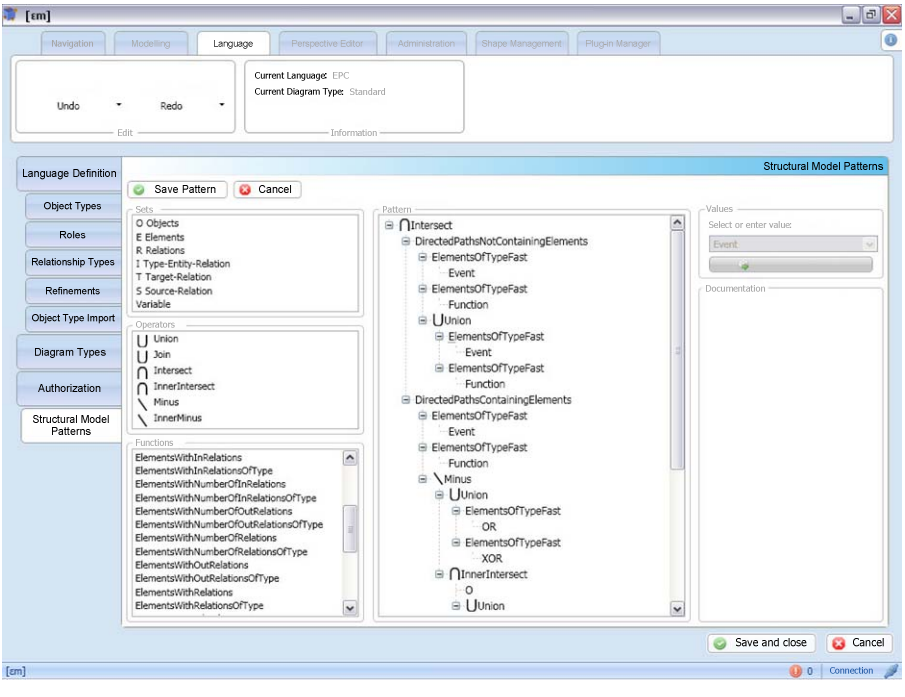

Fig. 11. Specification of the Pattern "Decision Split after Event" to Detect Errors in EPCs

The plug-in provides a specification environment for structural model patterns, which is integrated into the meta modeling environment of the tool, since the patterns are dependent on the respective modeling language. All basic sets, functions, and set operators introduced in Section 3 are provided and can be used to build up structural model patterns successively. In order to gain a better overview over the patterns, they are displayed and edited in a tree structure (cf. Fig. 11; here, the pattern example of Section 4 is shown).

The tree structure is built up through drag-and-drop of the basic sets, functions, and set operators. Whenever special characteristics of an according modeling language (e.g., *function*, *event*, *ET-RIRT*), numeric values, or names are used for the specification, this is expressed by using the "variable" element from the "sets" menu. The variable element, in turn, is instantiated by selecting a language-specific characteristic from the "values" 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 modeling language. Fig. 12 shows an exemplary model that was developed with the modeling language of EPCs and that contains a syntax error consisting of a decision split following an event. 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. 12).

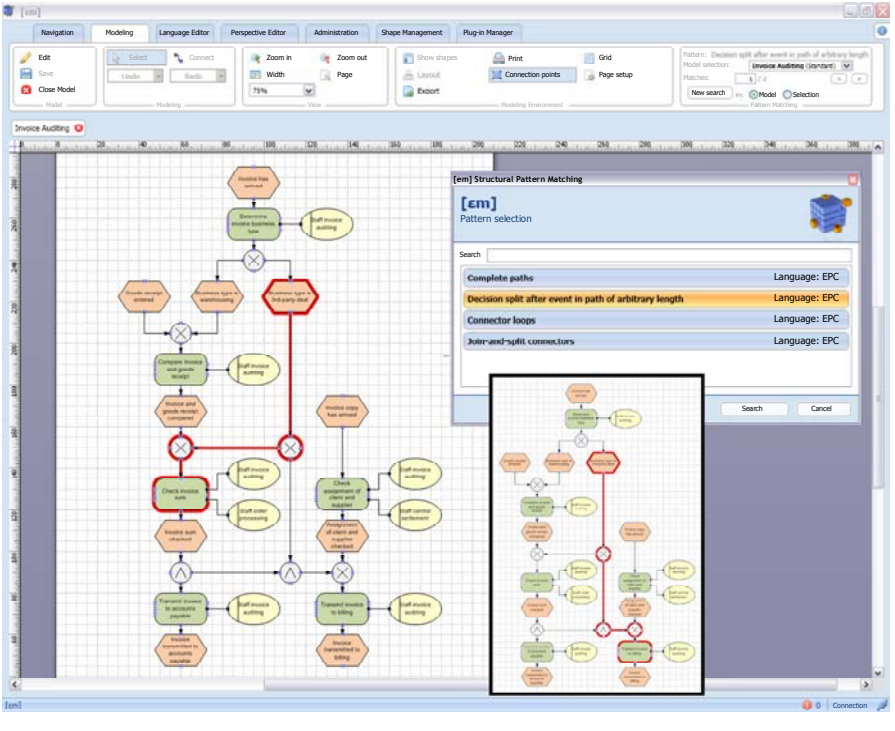

Fig. 12. Result of the Pattern Matching Process of "Decision Split after Event"

6 Conclusion and Outlook

Supporting model analysis by a generic pattern matching approach is promising, since it is not restricted to a particular problem area or modeling language. A first rudimentary evaluation through implementation and exemplary application of the approach has shown its general feasibility. Nevertheless, there still remains need for further research.

Hence, in the short term, we will focus on completing the evaluation of the presented approach. Although our current prototypical implementation already shows its general feasibility, further evaluation of our approach is necessary. 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 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. Although we strongly believe that our tool-implemented approach will inevitably support modelers in the task of model analysis, this needs to be objectively proven.

Medium-term research will address further applications for the structural model pattern matching approach presented here. For instance, we will question if modeling conventions on the basis of structural model patterns that are provided prior to modeling are able to increase the comparability of conceptual models.

References

1. Mendling, J.: Detection and Prediction of Errors in EPC Business Process Models. Doctoral Thesis, Vienna University of Economics and Business Administration (2007)
2. Vergidis, K., Tiwari, A., Majeed, B.: Business process analysis and optimization: beyond reengineering. IEEE Transactions on Systems, Man, and Cybernetics 38(1), 69–82 (2008)
3. Gori, M., Maggini, M., Sarti, L.: 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, Bath, pp. 81–88 (2005)
4. Scheer, A.-W.: ARIS – Business Process Modelling, 3rd edn., Berlin (2000)
5. Fu, J.: Pattern matching in directed graphs. In: Galil, Z., Ukkonen, E. (eds.) Proceedings of the 6th Annual Symposium on Combinatorial Pattern Matching, pp. 64–77. Espoo (1995)
6. Varró, G., Varró, D., Schürr, A.: 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, Brighton (2006)
7. Alexander, C., Ishikawa, S., Silverstein, M. A.: Pattern Language. New York (1977)
8. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design Patterns: Elements of Reusable Object-Oriented Software. New York (1995)
9. Fowler, M.: Patterns of Enterprise Application Architecture. Reading (2002)
10. van der Aalst, W.M.P., ter Hofstede, A.H.M., Kiepuszewski, B., Barros, A.P.: Workflow Patterns. Distributed and Parallel Databases 14(3), 5–51 (2003)
11. Rahm, E., Bernstein, P.A.: A survey of approaches to automatic schema matching. The VLDB Journal – The International Journal on Very Large Data Bases 10(4), 334–350 (2001)
12. Li, W., Clifton, C.: SemInt: a tool for identifying attribute correspondences in heterogeneous databases using neural network. Data & Knowledge Engineering 33(1), 49–84 (2000)

13. Madhavan, J., Bernstein, P.A., Rahm, E.: 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, Roma, pp. 49–58 (2001)
14. Aumueller, D., Do, H.-H., Massmann, S., Rahm, E.: 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 (2005)
15. Stumme, G., Mädche, A.: FCA-Merge: Bottom-up merging of ontologies. In: Nebel, B. (ed.) Proceedings of the 17thInternational Joint Conference on Artificial Intelligence, IJCAI 2001, August 4-10, 2001, pp. 225–230 (2001)
16. Shvaiko, P., Euzenat, J.: A Survey of Schema-Based Matching Approaches. In: Spaccapietra, S. (ed.) Journal on Data Semantics IV. LNCS, vol. 3730, pp. 146–171. Springer, Heidelberg (2005)
17. Hars, A.: Reference Data Models: Foundations of Efficient Data Modeling. In: German: Referenzdatenmodelle. Grundlagen effizienter Datenmodellierung, Wiesbaden (1994)
18. Chen, P.P.-S.: The Entity-Relationship Model: Toward a Unified View of Data. ACM Transactions on Database Systems 1(1), 9–36 (1976)
19. Hirschfeld, Y.: Petri nets and the equivalence problem. In: Börger, E., Gurevich, Y., Meinke, K. (eds.) Proceedings of the 7th Workshop on Computer Science Logic, Swansea, pp. 165–174 (1993)
20. de Medeiros, A.K.A., van der Aalst, W.M.P., Weijters, A.J.M.M.: Quantifying process equivalence based on observed behavior. Data & Knowledge Engineering 64(1), 55–74 (2008)
21. Hidders, J., Dumas, M., van der Aalst, W.M.P., ter Hofstede, A.H.M., Verelst, J.: When are two workflows the same? In: Atkinson, M., Dehne, F. (eds.) Proceedings of the 11th Australasian Symposium on Theory of Computing, pp. 3–11. Newcastle (2005)
22. van Dongen, B.F., Dijkman, R., Mendling, J.: Measuring similarity between business process models. In: Bellahsene, Z., Léonard, M. (eds.) Proceedings of the 20th International Conference on Advanced Information Systems Engineering, Montpellier, pp. 450–464 (2008)
23. Object Management Group (OMG): Meta Object Facility (MOF) Core Specification. Version 2.0 (2009), http://www.omg.org/spec/MOF/2.0/PDF
24. ISO: Concepts and Terminology for the conceptual Schema and the Information Base. Technical report ISO/TC97/SC5/WG3 (1982)
25. Object Management Group (OMG): Unified Modeling Language (OMG UML), Infrastructure, V2.1.2 (2009), http://www.omg.org/docs/formal/07-11-04.pdf
26. Delfmann, P., Knackstedt, R.: Towards Tool Support for Information Model Variant Management – A Design Science Approach. In: Österle, H., Schelp, J., Winter, R. (eds.) Proceedings of the 15th European Conference on Information Systems (ECIS 2007), St. Gallen, pp. 2098–2109 (2007)