Constructing a Bidirectional Transformation between BPMN and BPEL with a Functional Logic Programming Language

Steffen Mazanek[a,*], Michael Hanus[b]

[a] *Universität der Bundeswehr München, Germany*
[b] *Christian-Albrechts-Universität zu Kiel, Germany*

Abstract

In this article we show how functional logic programming techniques can be used to construct a bidirectional transformation between structured process models of the business process modeling notation (BPMN) and executable models of the business process execution language (BPEL). We specify the abstract syntax of structured process models by a context-free hypergraph grammar. This grammar can be subsequently transformed into a graph parser using our previously developed GRAPPA framework of functional logic GRAPh PArser combinators. The GRAPPA framework has been implemented using the functional logic programming language Curry. Furthermore, we show how the constructed parsers can be enriched with semantic computations as required for the synthesis of BPEL from BPMN. Since our parser is a function implemented in a functional *logic* language, it can be applied in both directions. Thus, given a BPEL model, a corresponding BPMN graph can be constructed with the very same parser. Finally, logic-based parsers can be used for model completion and language generation in a straightforward way.

In order to be self-contained, this article also surveys context-free hypergraph grammars, the concepts of the programming language Curry, the example languages BPMN and BPEL, and the ideas of the GRAPPA framework. Actually, this article is a literate Curry program and, as such, directly executable. Thus, it contains the complete concise source code of our application.

Keywords: graph parsing, functional logic programming, parser combinators, Curry, business process models, BPMN, BPEL

*Corresponding author

Email addresses: `steffen.mazanek@unibw.de` (Steffen Mazanek), `mh@informatik.uni-kiel.de` (Michael Hanus)

1. Introduction

This article describes how certain model transformations can be implemented in a convenient way using functional *logic* programming techniques [3]. The resulting transformations are not only very concise, but they also offer several unique features. Most importantly, they are bidirectional and provide syntactical model completion facilities for free.

As a running example, the transformation between process models of the Business Process Modeling Notation (BPMN) [49] and executable models of the Business Process Execution Language (BPEL) [48] is developed. Both languages are practically relevant and so is the transformation in-between. The key challenge for this transformation is that BPMN and BPEL belong to two fundamentally different classes of languages. BPMN is graph-oriented—and graphs actually will be used as an abstract representation [46]—while BPEL is mainly block-structured [51].

The proposed implementation of the transformation between BPMN and BPEL resorts to a framework of functional logic GRAPh PArser combinators, called Grappa [40]. For a long time, functional languages have already been known to be exceptionally well-suited for building *string* parsers—and parser combinators are probably the most popular approach in this respect: Just define some primitive parsers and combine them into advanced parsers via powerful combinators! Quite recently, the parser combinator approach has been lifted to the graph level. However, in the domain of graph parsing, features of logic programming languages appear to be very handy as well [40] so that functional logic languages [3] are promising in this area.

The Grappa framework can be used to construct graph parsers in a flexible manner. Indeed, parsers for several languages have already been implemented that way. However, there is one graph grammar formalism, namely Hyperedge Replacement Grammars (HRGs) [11], where the mapping between a grammar and its corresponding Grappa parser is particularly close. Considering the definition of HRGs and their properties, HRGs are the graph grammar formalism closest to context-free string grammars. Fortunately, a significant subset of BPMN can be described by an HRG and, thus, mapped to a parser in a straightforward way.

This article aims at being self-contained. As this introduction already shows, some preliminary concepts need to be introduced and discussed in order to present the actual parser in an understandable way. Hence, this article is necessarily multifaceted. However, the resulting transformation is worth the effort. It is very concise and readable so that this article can even contain the complete program code. Actually, the LaTeX source of this article is a literate program [33] and, as such, directly executable.

Note that the transformation presented in this article has been developed as a solution to the synthesis case [12] of the Graph-Based Tools (GraBaTs)

contest 2009.[1] Hence, the transformation part as such is not new at all. Actually, BPEL can be generated by most state-of-the-art business modeling tools. However, the proposed technique is novel and, where applicable, quite beneficial. The proposed solution also meets several of the evaluation criteria provided in the case definition [12], most prominently readability and reversibility—two important criteria hardly met by most other approaches.

This article is structured as follows: We start in the next section with an introduction of the languages BPMN and BPEL. Section 3 motivates hypergraphs as a uniform diagram representation model followed by a description of hyperedge replacement grammars that we use for the definition of the subset of BPMN covered in this article. Since our transformation approach is based on functional logic graph parser combinators, we review the necessary notions and techniques in the subsequent two sections. Section 4 introduces the basic concepts of the functional logic programming language Curry that are necessary to understand the subsequent program code. Also, (string) parser combinators are briefly reviewed in Section 4. Section 5 presents the Grappa framework of graph parser combinators, which is the basis of our transformation, and extends the original framework proposed in [40] by typed edges. We also show the close correspondence between grammar rules and parsers. The actual transformation between BPMN and BPEL is presented in Section 6. This transformation and the discussion of the overall transformation approach given in Section 7 are the main contributions of this paper. Finally, Section 8 reviews related work before we conclude in Section 9.

2. Source and Target Languages of the Example Transformation

In this section we briefly introduce the source and target languages of our example transformation, namely BPMN and BPEL, and discuss the challenges of this transformation task.

2.1. The Business Process Modeling Notation

The BPMN [49] has been established as the standard language for business modeling. As such it is widely adopted among business analysts and system architects. BPMN essentially provides a graphical notation for business process modeling with an emphasis on control flow. Models of the BPMN are basically flowcharts incorporating constructs tailored to business process modeling like different kinds of splits or events.

In the following, a core subset of BPMN elements that covers the fundamental kinds of control flow is considered. These elements are shown in Figure 1. BPMN is a graph-like language so that there are node objects and edges. The most important kind of edges are the sequence flow arrows that link two objects at a time and, thus, determine the control flow relation, i.e., the execution order. An object can be an activity, an event, or a gateway. Activities represent

[1] `http://is.ieis.tue.nl/staff/pvgorp/events/grabats2009/` (accessed on 2010-07-30)

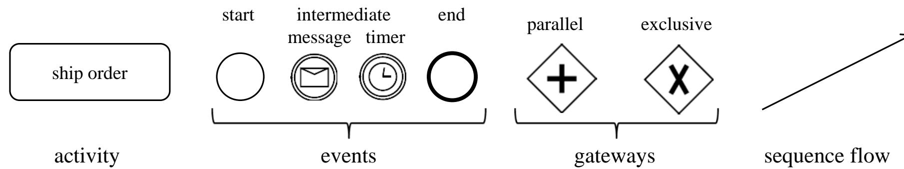


Figure 1: The core BPMN elements covered in this article

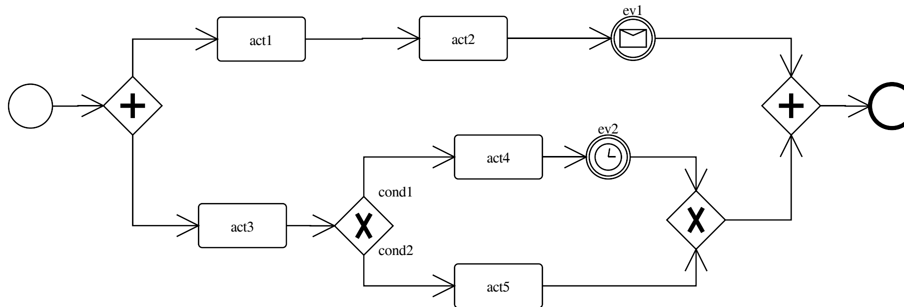


Figure 2: Example process

the atomic items of work to be performed. An event may signal the start of a process, the end of a process, a specific time that is reached during a process, or a message to be received.[2]

A gateway is a routing construct used to control the divergence and convergence of sequence flow. The BPMN distinguishes between

- parallel fork and join gateways for creating and synchronizing concurrent sequence flows, respectively,
- data/event-based exclusive decision gateways for selecting one out of a set of mutually exclusive alternative sequence flows where the choice is based on either the process data or external events (deferred choice, not considered in the following), and the corresponding gateways for joining them into one sequence flow again,
- some other kinds of gateways that we do not consider in the following.

Figure 2 shows an example process. It contains several activities and different kinds of intermediate events. Furthermore, it comprises both a parallel and an exclusive branching. This process is *well-structured*, i.e., splits and joins are properly nested such that each split has a corresponding join, and the process can be decomposed into so-called single-entry single-exit regions (SESE) regarding the sequence flow. This will become clearer in Section 3, where a precise

[2]Since messages can only be exchanged between different pools, which are not covered in this article, we do not consider them for our transformation.

grammar-based specification is given. Note that it is considered good practice to avoid arbitrary sequence flow—as in programming where spaghetti code is considered harmful. So, modelers should aim at creating structured processes. This restriction indeed is supposed to improve the quality and readability of process models [19, 44]. The transformation described in this article requires process models to be well-structured. A general discussion of the scope of our approach is provided in Section 7.

2.2. The Business Process Execution Language

BPEL [48] is essentially an imperative programming language targeted at web service implementations. Therefore, it is also called WS-BPEL. The syntax of the language is defined by an XML schema which is visualized by the class diagram shown in Figure 3(a). A BPEL process definition relates a number of activities. An activity is either a basic or a structured activity. Basic activities correspond to atomic actions such as invoking an operation `<invoke>`, waiting for a message `<receive>` or a particular amount of time `<wait>`, terminating the entire process `<exit>`, or doing nothing `<empty>`. Furthermore, we consider the following structured activities:

- sequence for defining an execution order `<sequence>`
- flow for parallel routing `<flow>`
- switch for conditional routing `<switch>`

There exist also other kinds of activities, like `<pick>` for dealing with race conditions, `<while>` for structured iteration, or `<scope>` for grouping activities, but they are not considered in the following.

A BPEL representation of the example process given in Figure 2 is shown in Figure 3(b). It clearly preserves the structure of the process. However, it is not the only possible representation as will be described in the next subsection.

Due to the practical relevance of BPEL for service orchestration, a lot of execution engines that support BPEL have emerged. Some of them even come with an associated graphical editing tool. However, the notation supported by these tools often directly reflects the underlying code, i.e., users are forced to think in terms of BPEL constructs. But the BPEL language is too restricted for analysts and, thus, unsuitable for the creative task of process modeling. They rather prefer BPMN, which gives much more freedom while modeling. This observation justifies the need for a mapping between BPMN and BPEL.

2.3. Transformation Challenges and Approaches

One of the key issues of the BPMN to BPEL transformation is that complex patterns in the input model need to be identified—in particular the SESE regions. The problem of identifying all SESE regions in a graph is a well-understood problem [29] which can be solved in linear time. Some existing transformations directly follow this approach for the construction of the so-called process structure tree [64]. Although such algorithms are available, it

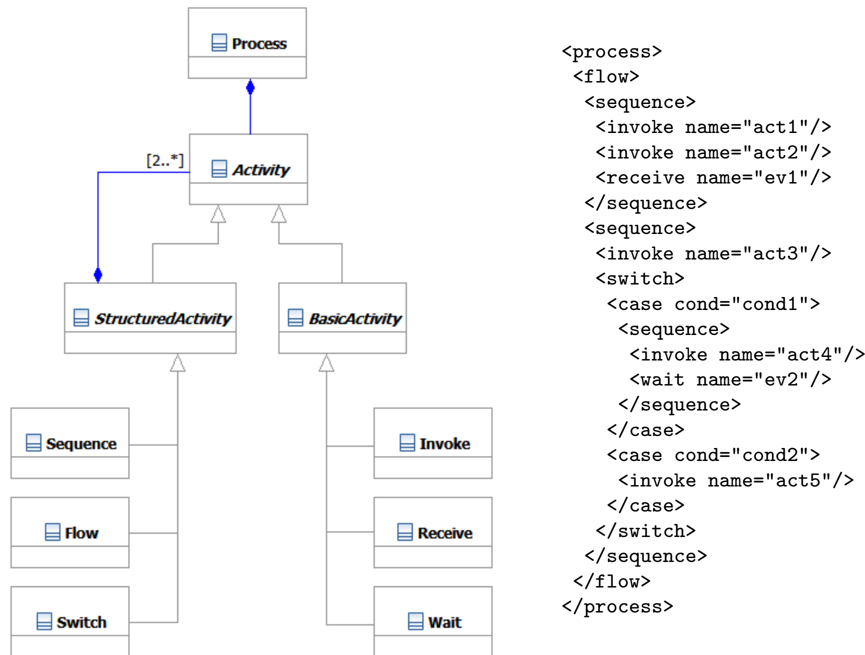


Figure 3: (a) BPEL syntax (left) and (b) example process corresponding to Figure 2 (right)

is worthwhile to see how this transformation can be solved with model transformation approaches in order to raise the level of abstraction and to improve readability. Note that, from an algorithmic point of view, the reverse transformation from BPEL to BPMN is the easier one, because in BPEL the structure is explicitly given by the tree structure of XML; it can be simply traversed.

Besides the identification of SESE regions, there is also a different way to derive BPEL from BPMN. BPEL provides a non-structured construct, so-called control links, that allows the definition of directed dependency graphs. A control link between activities A and B indicates that B cannot start before A has been completed. Also a condition can be attached that has to be true in order to execute B. There are some restrictions on the use of control links, e.g., they must not create cyclic control dependencies and must not cross the boundary of a while activity. Still a straightforward transformation from BPMN into BPEL can be implemented by excessively using control links and so-called event handlers [50]. However, this decreases readability since the structure of the process does not become as apparent as, e.g., in Figure 3(b).

A wide range of process models can be translated into BPEL respecting their structure as far as possible. For instance, an efficient transformation for quasi-structured processes, i.e., processes that can be re-written into perfectly structured ones [51], has been proposed that is based on SPQR tree decomposition [15]. However, this result is specific to the language and based on a manual programming effort.

In the GraBaTs 2009 synthesis case definition [12], four criteria for the evaluation of transformations have been proposed which can be used to evaluate the transformation approach presented in this article:

- Completeness: Which classes of processes can be transformed?
- Correctness: The transformation should preserve the execution semantics of the BPMN process model. This question is beyond the scope of this article which is mainly about syntax analysis. However, we will discuss the correctness of the proposed parser.
- Readability: The BPEL process definitions produced by the transformation as well as the transformation itself should be readable (although the latter is not explicitly stated in [12]). Readable BPEL basically means that block-structured control-flow constructs such as `<sequence>`, `<while>`, `<switch>`, etc. should be preferred over control links.
- Reversibility: The transformation should be accompanied by a reverse transformation; ideally a mechanism for bidirectional model synchronization should be provided.

3. Hypergraph Models

In this section we first introduce hypergraphs as a uniform diagram representation model [46]. We continue with the introduction of hypergraph grammars as a device for language definition. Finally, a common way for computing semantic representations of diagrams is described, namely attribute hypergraph grammars. That way, the introduction of the parser with semantic computations in Section 6 is prepared.

3.1. Hypergraphs as a Uniform Diagram Representation Model

Roughly speaking, *hypergraphs* are graphs where edges are allowed to connect (also called visit) an arbitrary number of nodes depending on the edges's labels (see [11] for formal definitions). In this case, the edges are called *hyperedges*. Vice versa, normal graphs are specific hypergraphs where all edges visit exactly two (not necessarily distinct) nodes, i.e., a source and a target node. For the sake of brevity, hypergraphs and hyperedges are occasionally just called graphs and edges in the following.

Figure 4 shows an example of a hypergraph, which actually corresponds to the process of Figure 2.[3] Hyperedges are graphically represented by boxes. Nodes are represented as black dots sometimes identified by numbers. The nodes visited by an edge are connected to this edge by lines called tentacles. Tentacles are numbered to indicate the roles of the connected nodes. For instance, BPMN

[3] Note that properties such as the labels of activities or the conditions of gateways are not graphically represented but can be attached as properties to the hyperedges.

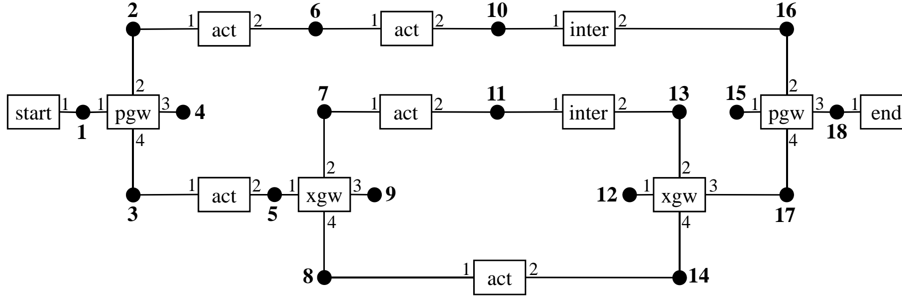


Figure 4: Hypergraph model of Figure 2

activities (edge label *act*) always have an incoming (1) and an outgoing tentacle (2); parallel and exclusive gateways (edge label *pgw*/*xgw*) always have four tentacles: left (1), top (2), right (3), and bottom (4).[4] Note that BPMN arrows for sequence flow are represented only implicitly in the hypergraph model shown in Figure 4. Thus, if two activities are connected by an arrow, the outgoing node of the first activity's edge coincides with the incoming node of the latter.

Hypergraphs are well-suited as a diagram model, because they can easily handle the requirement that diagram components typically have several distinct attachment areas at which they can interact with other diagram components. Thus, hypergraphs allow for a homogeneous representation of diagram components and their relations via hyperedges. Whereas the actual components can be modeled by hyperedges, attachment areas can be modeled by nodes visited by the hyperedge. This observation actually has led to the implementation of the diagram editor generator DiaGen [47], which is based on hypergraphs, hypergraph transformations, and hypergraph grammars.

Note that the transformation of a concrete business process diagram into its hypergraph representation is beyond the scope of this article. This actually is an easy task compared to the transformation of BPMN into BPEL, where the overall structure of the process needs to be identified. For instance, this step could be performed with a model transformation tool as an exogenous transformation. Layout information thereby can be completely discarded. However, then the reverse transformation would have to include a layout step for the concrete arrangement of the components. A DiaGen solution to the BPMN-to-BPEL case that includes an editor for business process diagrams has also been developed [43]. However, this solution uses another technique that will be briefly described in Section 3.3.

[4] Note that only three of those are used at a time, but with four tentacles the correspondence to the corners of the concrete diamond-shaped diagram component becomes more obvious.

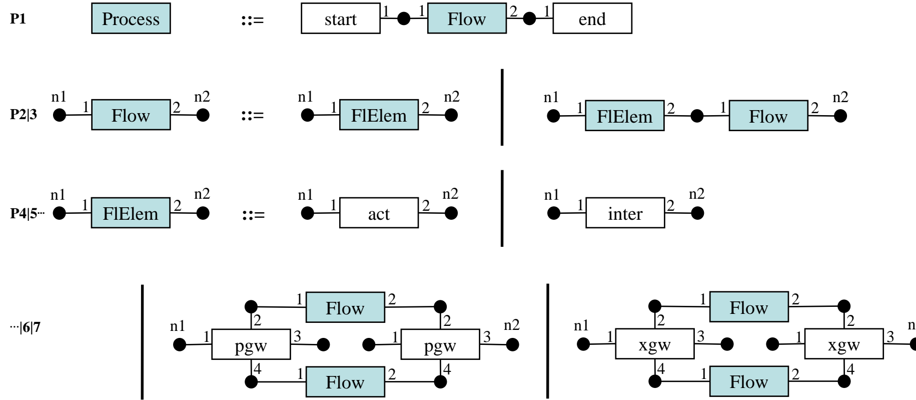


Figure 5: Productions of $G_{Process}$

3.2. Hypergraph Grammars for Language Definition

As already mentioned, the hypergraph language of well-structured process models can be defined using a Hyperedge Replacement Grammar (HRG) [11]. Each HRG consists of two finite sets of terminal and nonterminal hyperedge labels and a starting hypergraph, which contains only a single hyperedge with a nonterminal label. The syntax is specified by a set of hypergraph productions. Figure 5 shows the productions of the HRG $G_{Process}$. For the sake of conciseness, loops are not considered. Productions $L ::= R_1, \ldots, L ::= R_k$ with the same left-hand side L are drawn as $L ::= R_1 | \ldots | R_k$. The types *start*, *end*, *inter*, *act*, *pgw* and *xgw* are the terminal hyperedge labels. The set of nonterminal labels consists of *Process*, *Flow* and *FlElem*. The starting hypergraph consists of just a single *Process* edge.

The application of a context-free production removes an occurrence e of the left-hand side hyperedge from the host graph and replaces it by the hypergraph of the right-hand side, see [11]. Matching node labels of left- and right-hand sides determine how the right-hand side has to fit in after removing e. An example derivation is shown in Figure 6 (tentacle numbers are omitted in this figure). The hypergraph language generated by a grammar is defined by the set of all hypergraphs that can be derived from the starting hypergraph and that have only terminal edges. For $G_{Process}$ this is just the set of structured BPMN hypergraphs we want to cover.

As in the string setting, hypergraphs can be syntactically analyzed by a parser with respect to a given grammar. To this end, derivation trees are constructed (if existing). Such a hypergraph parser is part of the DiaGen system [47]. It uses dynamic programming techniques for the construction of derivation trees (similar to the string parser developed by Cocke, Younger, and Kasami [31]). The details of this parser are described in [5].

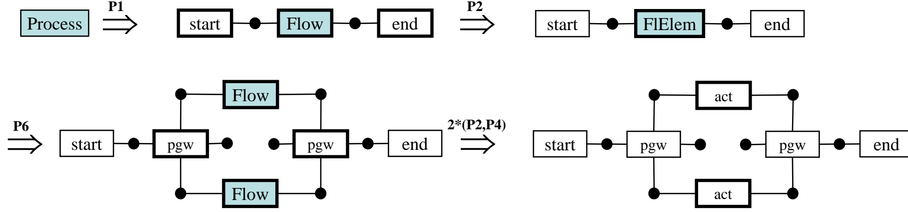


Figure 6: An example derivation

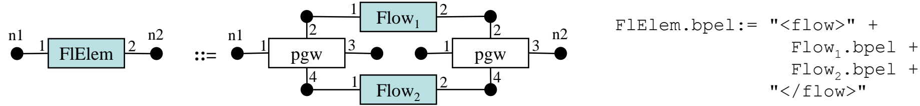


Figure 7: Example production with attribute evaluation rule

3.3. Attribute Hypergraph Grammars for Semantic Computations

The DiaGen system also supports the translation of a diagram into some domain-specific data structure. For this purpose, it provides attribute evaluation [45], a well-known technique in the domain of compiler construction [32]. Basically, each hyperedge carries some attributes. Number and types of these attributes depend on the hyperedge label. Productions of the hypergraph grammar may be augmented by attribute evaluation rules which assign values to attributes of the corresponding edges. An attribute hypergraph grammar for the synthesis of BPEL from BPMN has been proposed in [43] as an alternative solution to the case. An example production with an attribute evaluation rule is shown in Figure 7. The subscript numbers do not belong to the edge label but are used to refer to the different occurrences of edges with the same label.

After parsing, attribute evaluation works as follows. Each hyperedge that occurs in the derivation tree has a distinct number of attributes. Grammar productions that have been used for creating the tree impose instantiated attribute evaluation rules. Those determine how attribute values are computed as soon as the values of others are known. The attribute evaluation mechanism then computes a valid evaluation order. Some or even all attribute values of terminal edges are already known; they have been derived from attributes of the diagram components. For instance, hyperedges with label *act* have a `name` attribute representing the name of the respective activity.

Attribute string grammars can be translated into parsers in a straightforward way, e.g., by using a parser generator or a combinator framework. This also applies to the graph parser combinators that will be introduced in Section 5. A disadvantage of attribute grammars is, however, that usually the reverse transformation cannot be derived automatically (pair grammars have been suggested to this end [53]).

4. Functional Logic Programming and Parser Combinators

Since our framework of graph parser combinators is implemented in the programming language Curry [22], we introduce the relevant concepts of Curry in this section. Moreover, we discuss the ideas behind (string) parser combinators and recapitulate why functional logic languages are a natural platform for those.

4.1. The Programming Language Curry

Curry is a declarative multi-paradigm language combining features from both functional and logic programming (see [3, 24] for recent surveys). The syntax of Curry is close to Haskell [52]. The main addition are free (logic) variables in conditions and right-hand sides of defining rules. In functional programming, one is interested in the computed value, whereas logic programming emphasizes the different bindings (answers). Consequently, Curry computes for each input expression its value together with a substitution for the free variables occurring in the input expression.

A Curry program consists of the definition of data types and operations on these types. Although we often use the term "function" as a synonym of "defined operation" as in functional programming, it should be noted that operations in Curry might yield more than one result on the same input due to the logic programming features. Thus, Curry operations are not functions in the classical mathematical sense so that some authors use the term "non-deterministic functions" [17]. Nevertheless, Curry programs have a purely declarative semantics where non-deterministic operations are modeled as set-valued functions (to be more precise, down-closed partially ordered sets are used as target domains in order to cover non-strictness, see [17] for a detailed account of this model-theoretic semantics).

In a Curry program, operations are defined by conditional equations with constraints in the conditions. They are evaluated lazily and can be called with partially instantiated arguments. This feature will be heavily used later on. Calls with free variables are evaluated by a possibly non-deterministic instantiation of the required arguments, i.e., arguments whose values are demanded to decide the applicability of a rule. This mechanism is called *narrowing* [58]. Curry is based on a refinement of narrowing, called "needed narrowing" [1], which is optimal for particular classes of programs, i.e., shortest derivations and a minimal number of solutions are computed (see [1] for more details).

As an example, consider the Curry program given in Figure 8. It defines a polymorphic data type for lists, and operations to compute the concatenation of lists and the last element of a list. Note, however, that lists are a built-in data type with a more convenient syntax, e.g., one can write `[x,y,z]` instead of `x:y:z:[]` and `[a]` instead of the list type "`List a`".

Logic programming is supported by admitting function calls with free variables, see `(ys++[z])` in Figure 8, and constraints in the condition of a defining rule. In contrast to Prolog, free variables need to be declared explicitly to make their scopes clear; in the example this is expressed by "`where ys,z free`".

```
data List a = [] | a : List a    --[a] denotes "List a"

--"++" is a right-associative infix operator
(++) :: [a] → [a] → [a]
[]     ++ ys = ys
(x:xs) ++ ys = x : (xs ++ ys)

last :: [a] → a
last xs | (ys ++ [z]) =:= xs
        = z                    where ys,z free
```

Figure 8: A Curry program with operations on lists

Conditional program rules have the form "l | c = r" specifying that l is reducible to r if c is satisfied. The condition c is a constraint, i.e., an expression of the built-in type `Success`. For instance, the trivial constraint `success` is an expression of type `Success` that denotes the always satisfiable constraint. An equational constraint e_1 `=:=` e_2 is satisfiable if both sides e_1 and e_2 are reducible to unifiable constructor terms. Furthermore, if c_1 and c_2 are constraints, c_1 `&` c_2 denotes their concurrent conjunction, i.e., this expression is evaluated by proving both argument constraints concurrently. For instance, the rule defining `last` in Figure 8 states in its condition that `z` is the last element of a given list `xs` if there exists a list `ys` such that the concatenation of `ys` and the one-element list `[z]` is equal to the given list `xs`.

Note that, in contrast to Haskell, the evaluation of "`last []`" does not raise a run-time error but *fails*. Similar to logic programming, failure is not a run-time error in Curry but a useful programming technique: if some evaluation, i.e., a rule application, leads to a failure, a different rule is chosen. Thus, *all* applicable rules are tried to evaluate some function call (in contrast to Haskell [52] which commits to the first applicable rule). This kind of non-determinism can be implemented by backtracking or by a parallel search for all solutions. The concrete implementation is specific to each Curry system.

The execution model of Curry is based on narrowing, a combination of variable instantiation and reduction. For instance, the expression `(ys++[1])` is evaluated to several results: the list `[1]` by instantiating `ys` to `[]` using the first equation, to the partially unknown list `[x,1]` by instantiating `ys` to `[x]` using the second equation and then the first equation, and so on to terms of the form `[x,y,1]`, `[x,y,z,1]`, ... To restrict these potentially infinite number of non-deterministic computations, one usually puts such expressions in the context of constraints. For instance, the equational constraint `(ys++[x])=:=[1,2]` causes a finite computation space containing the unique solution where `ys` and `x` are bound to `[1]` and `2`, respectively. This technique is exploited in Figure 8 to define the operation `last` conveniently.

As already remarked, operations in Curry may yield more than one result for some input due to the logic programming features. For instance, consider

the operation `removeSome` defined as follows:

```
removeSome :: a → [a] → [a]
removeSome x xs | xs=:=(xs1++x:xs2) = xs1++xs2
                where xs1,xs2 free
```

The condition states that the input list `xs` is decomposable into `xs1`, `x`, and `xs2`, and the result is the concatenation of `xs1` and `xs2`. Thus, `removeSome` returns *any* list which can be obtained by removing some occurrence of the input element `x` from the input list `xs`. Actually, *non-deterministic operations* like `removeSome` are a useful programming technique which will be used to implement (graph) parsers in a convenient manner.

The combination of functional and logic programming features has led to new design patterns [2] and better abstractions for application programming, e.g., as shown for programming with databases [7, 13], GUI programming [20], web programming [21, 23, 26], or string parsing ([10] and Section 4.2). In this article, we show how to exploit these combined features to construct a bidirectional transformation between a visual graph-oriented language and a term-oriented language.

As already mentioned, this article is a literate Curry program [33], i.e., the article's source text is directly executable. In a literate Curry program, all real program code starts with the special character "`>`". This is called Bird-style code and comments. Curry code not starting with "`>`", e.g., the code in Figure 8, is just for reader's information and not required to run the program, i.e., it is a comment. Occasionally, we will also use Curry line comments, which start with "`--`". As an example of code that counts, consider the following declaration of a `main` operation which applies the parser `processS` (to be defined in Section 6) to the example hypergraph `ex` shown in Figure 4 and pretty-prints the resulting BPEL representation in XML:

```
> main = let bpel = getSemRep (findfirst (\s → s=:=processS ex))
>         in putStrLn (bpel2xml bpel)  --output result
```

Since operations in Curry might yield more than one result, we select one value of "`processS ex`" using `findfirst` and print its XML representation (the operations used here will be defined later). Thus, executing this program will result in the output shown in Figure 3(b).

There exist various implementations of Curry,[5] where three major systems are available for general use and, thus, can run our transformation:

- MCC (Münster Curry Compiler) [35] generates fast native code.
- PAKCS (Portland Aachen Kiel Curry System) [25] compiles Curry to Prolog as an intermediate language.
- KiCS (Kiel Curry System) [8] compiles Curry to Haskell.

[5] `http://www.curry-language.org/implementations/overview` (accessed on 2010-07-30)

In functional programming, the most popular approach to parsing is based on *parser combinators*. Following this approach, some primitive parsers are defined that are combined into more advanced parsers using a set of powerful combinators. These combinators are higher-order functions that can be used to define parsers so that they resemble the corresponding grammar very closely [28].

Parser combinators integrate seamlessly into the rest of the program, hence the full power of the host language, e.g., Haskell [52] or Curry, can be used. Unlike parser generators such as Yacc [30], no extra formalism is needed to specify a grammar. It is rather possible to define a powerful domain-specific *embedded* language for the convenient construction of parsers. Another benefit of the combinator approach is that parsers are first-class values within the language. For example, lists of parsers can be constructed, and parsers can also be passed as parameters. The possibilities are only restricted by the potential of the host language.

In most purely functional parser combinator approaches, a parser is a function that takes a sequence of tokens—plain characters in the simplest case—as input and returns a list of successes [65] containing all possible ways in which a *prefix of the input* can be recognized and converted into a value of the result type. This parser result could be, e.g., a derivation tree or a computed value similar to attribute grammars [32]. Each result is combined with the corresponding remaining part of the input. This is important in order to compose parsers sequentially afterwards. Indeed, the subsequent parser needs to know where it has to continue the analysis.

Combinators such as sequence (denoted by the `*>` operator in the following) or choice (`<|>`) are higher-order functions, i.e., functions whose parameters are functions again. For instance, consider a function `symb` with type `Char` → `Parser` that constructs a parser accepting a particular character. Then a list `pl` of parsers can easily be constructed, e.g., by applying `symb` to each letter:[6]

```
pl = map symb ['a'..'z']
```

A parser `lcl` that accepts an arbitrary lower-case letter then can be implemented by folding `pl` via the choice operator and the never succeeding parser `fail` as the neutral element:[7]

```
lcl = foldr (<|>) fail pl
```

With the use of sequence and choice combinators, any context-free string grammars can directly be mapped to a parser.[8] For instance, consider the grammar

[6] The standard function `map :: (a → b) → [a] → [b]` applies a function to each element of a list [52, 22].

[7] The standard function `foldr :: (a → b → b) → b → [a] → b` folds a list by using a function applied from right to left in the list [52, 22].

[8] Several systems require the elimination of left-recursive rules, but this is always possible.

$$
\begin{array}{rcl}
B & ::= & \text{`('}\ B\ \text{`)'}\ B \\
  & | & \epsilon
\end{array}
$$

of well-formed bracket terms derivable from the only nonterminal and, thus, starting symbol B. Given an always succeeding primitive parser `epsilon` that does not consume any input, a parser for this language can be constructed as follows:

```
b =     symb '(' *> b *> symb ')' *> b
    <|> epsilon
```

A lot of research has been put into developing more and more sophisticated string parser combinator libraries, e.g., [34, 61, 66]. Those rely on different approaches to search, backtracking control, parser composition, and result construction.

Functional *logic* parser combinators, proposed for the first time in [10], have all the advantages of purely functional ones but additionally benefit from the following merits of logic-based parsers:

- Non-determinism is the default behavior, i.e., it is not necessary to implement search based on backtracking or the "lists of successes" technique [65] by hand.
- Parsers can be used for language generation thanks to the fact that a parser is not a function but rather a relation.
- Incomplete information can be handled conveniently by using free (logic) variables.

The possibilities of dealing with incomplete information are particularly interesting. For instance, free variables can be used as placeholders in the input sentence. While parsing, these variables are instantiated with the tokens that can occur at this particular position in order to get a valid sentence of the language. In the string setting, this is a nice feature coming for free but not being useful directly. For instance, to exploit this for error correction, the position and number of the missing token needs to be known or all possible positions would have to be tried. This is not really feasible. However, in the graph setting, where we deal with sets, this effect is much more powerful as we will see. Actually, this is already the key idea for deriving meaningful graph completions.

Note that purely logic languages would not be equally well-suited for our purpose, because those do not support the straightforward definition of higher-order functions such as combinators. Thus, the "remaining input" would have to be passed more explicitly resulting in a lot of boilerplate code, i.e., code that has to be included in many places of the program with little or no alteration. Prolog provides *Definite Clause Grammars* (DCGs), a syntactic sugar to hide the *difference list* mechanism needed to build reasonably efficient string parsers

in logic languages. However, a graph is not linearly structured. Hence, this notation cannot be used for graph parsing. Tanaka's *definite clause set grammars* [63], a DCG-like formalism for free-word-order languages, are not supported by common Prolog systems.

Having this in mind, graph parsing appears to be a domain asking for multi-paradigm declarative programming languages [24] such as Curry. In this domain their benefits really stand out. The inherent need for logic features in graph parsing will be further motivated in the following.

5. Graph Parser Combinators: The Grappa Framework

The graph parser combinator library Grappa allows the convenient and straightforward construction of hypergraph parsers for HRGs. The implementation of Grappa in Curry indeed exploits both functional and logic programming techniques. That way, the following features have been achieved:

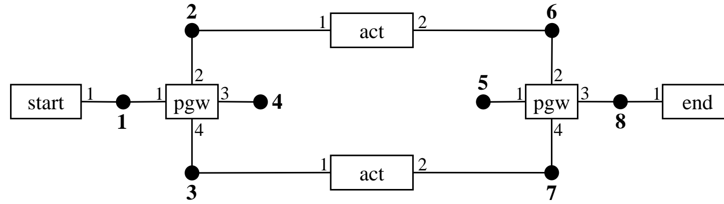
- Straightforward translation of HRGs to reasonably efficient parsers.
- Application-specific results due to powerful semantic computations.
- Easy to use for people familiar with parser combinators.
- Usable context information. This allows the convenient description of languages that cannot be defined with an HRG.
- Robust against errors. Valid subgraphs can be extracted.
- Bidirectionality. Besides syntax analysis, parsers can be used to construct or complete graphs of the intended language or even to create a graph from its semantic representation as required for the transformation of BPEL into BPMN.

In this section, we introduce all parts of the Grappa[9] library that are required for the implementation of the BPMN parser in Section 6.

5.1. Type Declarations for Graphs and Graph Parsers

We start by defining the basic data structures for representing graphs and hypergraphs. For the sake of simplicity, nodes are represented by integer numbers. A graph is declared as a list of edges each with its incident nodes. A type for the edge labels `t` can be passed as a type parameter (in the simplest case this can be just strings representing their labels as in the original Grappa version [40]). The tentacle numbers correspond to the position of a node in the list of incident nodes.

[9] `http://www.unibw.de/inf2/grappa` (accessed on 2010-07-30)




```
> ex_sm :: Graph String
> ex_sm = [("start", [1]), ("pgw", [1,2,4,3]), ("act", [2,6]),
>          ("act", [3,7]), ("pgw", [5,6,8,7]), ("end", [8])]
```

Figure 9: A small example BPMN hypergraph and a representation as a Curry term

```
> type Node = Int
> type Edge t = (t, [Node])
> type Graph t = [Edge t]
```

Note that the actual order of edges in the list does not matter. Rather the list of edges representing a graph is interpreted as a multiset. Consequently, there are a lot of terms describing the very same graph. This approach is the easiest one to implement and understand, but it has also some weaknesses. Those will be discussed in Section 7. Furthermore, isolated nodes cannot be directly represented, because the nodes of a hypergraph are implicitly given as the union of all nodes incident to its edges. But this restriction is not severe since isolated nodes simply do not occur in many hypergraph application areas. In the context of visual languages, diagram components are represented by hyperedges, and nodes just represent their attachment areas, i.e., each node is visited by at least one edge (see Section 3).

Figure 9 shows a small example of a BPMN hypergraph and its corresponding graph representation in Curry where `String`-labeled edges are used. Note that node numbers and tentacle numbers have to be clearly distinguished. The tentacle numbers are only represented implicitly as the position of a node in an edge's node list.

Next, the declaration of the type `Grappa` representing a graph parser is introduced. This type is parameterized over the type `res` of semantic values associated to graphs. Graph parsers are non-deterministic operations from graphs to pairs consisting of the parsing result and the graph that remains after successful parser application. Note that the non-remaining parts of the graph have been consumed in the course of parsing. In contrast to Haskell, it is not required to explicitly deal with parsing errors and backtracking. Instead, similar to [10], the non-determinism inherent to functional logic programming languages is exploited, which yields the following, very concise type declaration:

```
> type Grappa t res = Graph t  →  (res, Graph t)

> getSemRep (r,_) = r    --access semantic representation
```

5.2. Primitive Parsers and Basic Combinators

Now we define some primitives for the construction of graph parsers. Given an arbitrary value `v`, `pSucceed` always succeeds returning `v` as a result without any consumption of the input graph `g`:

```
> pSucceed :: res  →  Grappa t res
> pSucceed v g = (v, g)
```

In contrast, `eoi` only succeeds if the graph is already completely consumed. In this case, the result `()` is returned, the only value of the so-called unit type. Note that it is not necessary to state explicitly that `eoi` fails on non-empty inputs—the absence of a rule is sufficient.

```
> eoi :: Grappa t ()
> eoi [] = ((), [])
```

An especially important constructor of primitive parsers is `edge`, the graph equivalent of `symb`, cf. Section 4.2. The parser "`edge e`" only succeeds if the given edge `e` is part of the given input graph `g`. In contrast to `symb`, it does not have to be at the *beginning* of the token list though. If `g` contains `e`, `e` is consumed, i.e., removed from `g`. The parser `edge` is implemented in a logic programming style making use of an equational constraint (note the similarity to the operation `removeSome` in Section 4.1).

```
> edge :: Edge t  →  Grappa t ()
> edge e g | g=:=(g1++e:g2) = ((), g1++g2)
>            where g1, g2 free
```

If `g`, which is the list representation of the input graph, can be decomposed into `g1`, `e` and `g2`, the edge `e` indeed is contained in `g`. In this case, `e` has to be consumed. This is realized by returning just `g1++g2` as the remaining graph.

Figure 10 defines some basic graph parser combinators. The choice operator `<|>` takes two parsers and succeeds if either the first or the second one succeeds on the particular input graph `g` (note that the non-determinism provided by functional logic programming is quite handy here). Two parsers can also be combined via `<*>`, the successive application where the result is constructed by function application. Thereby, the first parser `p` has to return a function `pv` as result that eventually is applied to the result `qv` returned by the successive parser `q`. Note that `q` is applied to `g1`, i.e., the input that `p` has left. The correct (intended) order of evaluation is enforced by using Curry's `case` construct.

For convenience, we define the combinators `*>` and `<*` that throw away the result of either the left or the right parser. Indeed, the pure success of a parser is often sufficient for the remaining computation. Then a particular result is not at all required. For plain language recognition without semantic computations, only the combinator `*>` is needed, see the (textual) bracket term example of Section 4.2. Note that the result types of the argument parsers in the combinators `*>` and `<*` might differ but they have to process graphs with the same edge type `t`. This is reasonable since they operate on the same graph but can compute different kinds of semantic information.

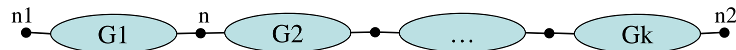
```
> (<|>) :: Grappa t res  → Grappa t res  → Grappa t res
> (p1 <|> p2) g = p1 g
> (p1 <|> p2) g = p2 g

> (<*>) :: Grappa t (r1 → r2)  →  Grappa t r1  →  Grappa t r2
> (p <*> q) g = case p g of
>                (pv, g1)  →  case q g1 of
>                             (qv, g2)  →  (pv qv, g2)

> (<$>) :: (res1 → res2)  →  Grappa t res1  →  Grappa t res2
> f <$> p = pSucceed f <*> p

> (<$)  :: res1  →  Grappa t res2  →  Grappa t res1
> f <$ p = const f <$> p

> (<*)  :: Grappa t res1  →  Grappa t res2  →  Grappa t res1
> p <* q = (\x _  →  x) <$> p <*> q

> (*>)  :: Grappa t res1  →  Grappa t res2  →  Grappa t res2
> p *> q = (\_ x  →  x) <$> p <*> q
```

Figure 10: Basic graph parser combinators

The parser transformers `<$>` and `<$` can be used to either apply a function to the result of a parser or to just replace it by another value, respectively. Note that some of the other combinators are defined in terms of `<$>`. For instance, `<$` can be defined using `<$>` by constructing a constant function from the respective value.[10] Another example application of `<$>` is given below in the definition of the `chain` combinator.

On top of these basic combinators, various other useful combinators can be defined. We show one example: The combinator `chain p (n1,n2)` can be used to identify a non-empty chain of graphs that can be parsed with `p`. This chain has to be anchored between the nodes `n1` and `n2`.

```
> chain::((Node,Node) → Grappa t a) → (Node,Node) → Grappa t [a]
> chain p (n1,n2) = (:[]) <$> p (n1,n2)
> chain p (n1,n2) = (:)   <$> p (n1,n) <*> chain p (n,n2)
>                   where n free
```



[10]Given a particular value v, `const` v returns a constant function that maps every argument to v [52, 22].

The definition of `chain` can be read as: "`chain p (n1,n2)`" succeeds if "`p (n1,n2)`" succeeds. Then a singleton list with its result is returned.[11] It may also succeed by running "`p (n1,n)`" for some node `n` and thereafter "`chain p (n,n2)`" again. Then their results are combined via the list constructor `(:)`. Note that `chain` can be conveniently defined because the inner node `n` does not need to be known in advance. It can simply be defined as a free variable which will be instantiated according to the narrowing semantics of Curry. Representing graph nodes as free variables actually is a functional logic design pattern [2] that we will use some more times in the remainder of this article.

5.3. Translation of the BPMN Grammar to a Parser

Before the BPMN to BPEL transformation will be introduced in Section 6, we show the implementation of a direct mapping from HRGs to Grappa parsers. We will see that parsers indeed resemble grammars very closely.

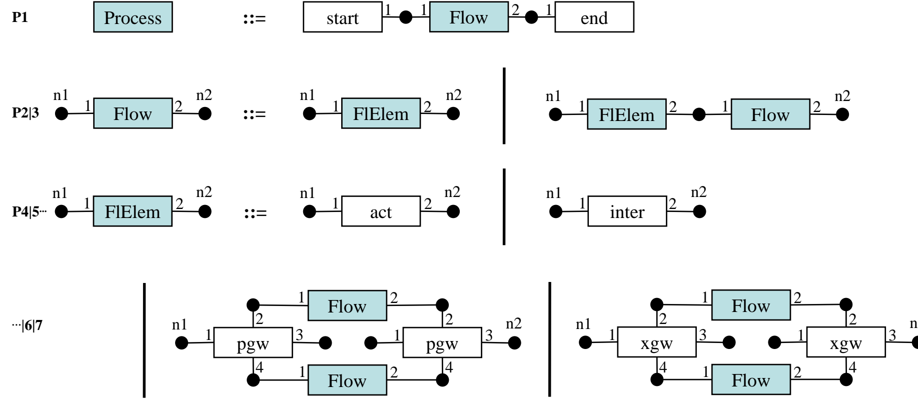
Figure 11 presents a plain parser for BPMN hypergraphs that does not perform any semantic computations, i.e., its result type is the unit type `()`. The type annotations are for convenience only and, thus, can just as well be omitted. For each nonterminal edge label *nt*, a parser function has to be defined that takes a tuple of nodes $(n_1, ..., n_k)$ compatible to the edge's label as a parameter. A new function body is inserted for each production over *nt* (alternatively `<|>` could be used). Each terminal edge in the right-hand side of the production is matched and consumed using the primitive parser `edge`, each nonterminal one is translated to a call of the function representing this nonterminal. Note that it would be also possible to define the parser in a more homogeneous way by introducing a special parser function for each terminal edge type, such as `act (n1,n2) = edge ("act", [n1,n2])`.

But what about the inner nodes within the right-hand side of a production? Nothing is known about these nodes yet. Therefore, as usual in logic programming, a free variable is introduced for each of them in the same manner as we have already seen in the definition of `chain`. Those nodes are instantiated during the process of parsing.

In contrast to string parsing, the order of parsers in a successive composition via `*>` is not that important as long as left recursion is avoided. Nevertheless, the chosen arrangement might have an impact on the performance. Usually, it is advisable to start with a terminal edge and continue by traversing the right-hand side graph along shared nodes.

Parsers defined in such a way are quite robust. For instance, they ignore redundant components, i.e., those just remain at the end. Still, complete input consumption can be enforced by a subsequent application of `eoi`. Thus, instead of `process`, the extended parser "`process <* eoi`" would have to be used. This parser can be called as follows:

[11] `(:[])` is the notation of Haskell or Curry for sections, i.e., if we apply `(:[])` to an element `x`, we get `(x:[])` which is identical to the one-element list `[x]`.




```
> process :: Grappa String ()
> process = edge ("start", [n1]) *>
>           flow (n1,n2) *>
>           edge ("end", [n2])
>           where n1,n2 free

> flow :: (Node,Node)  → Grappa String ()
> flow (n1,n2) = flElem (n1,n2)
> flow (n1,n2) = flElem (n1,n) *>
>                flow (n,n2)
>                where n free

> flElem :: (Node,Node)  → Grappa String ()
> flElem (n1,n2) = edge ("act", [n1,n2])
> flElem (n1,n2) = edge ("inter", [n1,n2])
> flElem (n1,n2) = edge ("pgw", [n1,n1t,n1r,n1b]) *>
>                  flow (n1t,n2t) *>
>                  edge ("pgw", [n2l,n2t,n2,n2b]) *>
>                  flow (n1b,n2b)
>                  where n1t,n1r,n1b,n2l,n2t,n2b free
> flElem (n1,n2) = edge ("xgw", [n1,n1t,n1r,n1b]) *>
>                  flow (n1t,n2t) *>
>                  edge ("xgw", [n2l,n2t,n2,n2b]) *>
>                  flow (n1b,n2b)
>                  where n1t,n1r,n1b,n2l,n2t,n2b free
```

Figure 11: Production of $G_{Process}$ (cf. Figure 5) and the corresponding parser implementation

```
BPMN2BPEL> (process <* eoi) ex_sm
((),[])
More solutions? [Y(es)/n(o)/a(ll)] y
No more solutions
```

So, for the example input graph `ex_sm` shown in Figure 9, the pair `((),[])` is returned. The list part `[]` represents the remaining graph, which is empty here (actually, we have enforced this with `eoi`). As a result the value `()` of unit type `()` is returned. We can conclude from this evaluation that the graph `ex_sm` is syntactically correct. Since we have searched for further solutions by pressing "y", but did not found any, we can also conclude that there is only a single derivation tree for `ex_sm` (the grammar $G_{Process}$ actually is unambiguous). However, we cannot tell anything about the structure and meaning of `ex_sm`. This gap will be closed in the following section.

6. Constructing the BPMN to BPEL Transformation

Our parser constructed so far can only check whether the given graph is, or at least contains, a valid BPMN hypergraph. However, a major benefit of the combinator approach is that language-specific results can be computed in a flexible way. In this section we show how one can synthesize BPEL from BPMN by adding this transformation as a semantic computation to the BPMN parsing process. In the subsequent section, we discuss how this implementation can be exploited for various tasks beyond the transformation of BPMN into BPEL.

6.1. Typed BPMN Hypergraphs

In order to transform BPMN into BPEL, edges of the BPMN hypergraphs have to carry attributes, e.g., semantic information describing the label of an activity or the kind of an intermediate event. For this purpose, we introduce the type of BPMN edges by the declaration of the data type `BPMNComp`:

```
> data BPMNComp =
>       BPMNStart | BPMNEnd | BPMNPGW |
>       BPMNXGW String String |        --conditions as params
>       BPMNActivity String |          --label as param
>       BPMNInter BPMNInterKind String --kind and label as params
>
> data BPMNInterKind = BPMNWait | BPMNReceive
```

In contrast to the original work on functional logic graph parser combinators [40], we have extended the Grappa framework to *typed* hypergraphs. This extension is important for the comprehensive and correct generation of the corresponding BPEL code. For instance, we can represent the example BPMN diagram of Figure 2 as a graph over the edge type `BPMNComp` by the constant `ex` given in Figure 12.

Actually, there are some degrees of freedom in modeling BPMN. For instance, we could have also represented the kind of gateway, i.e., exclusive or parallel, via an attribute (in the same way as we distinguished events).

```
> ex :: Graph BPMNComp
> ex = [(BPMNStart,[1]),(BPMNPGW,[1,2,4,3]),(BPMNActivity "act1",[2,6]),
>       (BPMNActivity "act2",[6,10]),(BPMNPGW,[15,16,18,17]),
>       (BPMNActivity "act3",[3,5]),(BPMNXGW "cond1" "cond2",[5,7,9,8]),
>       (BPMNActivity "act4",[7,11]),(BPMNActivity "act5",[8,14]),
>       (BPMNInter BPMNWait "ev2",[11,13]),(BPMNXGW "" "",[12,13,17,14]),
>       (BPMNInter BPMNReceive "ev1",[10,16]),(BPMNEnd,[18])]
```

Figure 12: Representation of the example process in Figure 2 as a Curry term

6.2. Representing BPEL by Constructor Terms

To implement the parser that transforms BPMN into BPEL, we have to define a representation of BPEL in Curry. The significant subset of BPEL introduced in Section 2.2 can be modeled as follows:

```
> type BPEL = [BPELComp]

> data BPELComp = Invoke String | Wait String | Receive String |
>                 Flow BPEL BPEL | Switch String String BPEL BPEL
```

Thus, an element of type `BPELComp` describes some activity of a BPEL process, and elements of type `BPEL` (occurring as arguments of structured activities) are simply sequences of BPEL activities. Since these sequences can be arbitrarily nested in structured activities, the target structure of our transformation is basically a tree (in contrast to the graph-based source structure of type `Graph BPMNComp`). Note that terms of the tree type `BPEL`[12] can be transformed into BPEL-XML in a straightforward way as shown in Appendix A. One could also directly construct XML while parsing. However, the language Curry allows for more flexibility when dealing with constructor terms as above. This will be important for supporting the reverse transformation.

6.3. Adding Semantics to the BPMN Parser

Figure 13 shows the main part of our implementation, i.e., the extended parser for BPMN where semantic computations have been added. As can be seen by the type of the main parsing operation `processS`, this parser works on typed hypergraphs with edges of type `BPMNComp` and returns a semantic value of type `BPEL` (the BPEL tree corresponding to the input BPMN graph). To distinguish this parser from the plain parser without semantic computations (result type `()`) presented in Figure 11, each function name is suffixed with the capital letter `S`. Note that both parsers accept exactly the same graph language (apart from the different attributes associated to edges), i.e., the language generated by the grammar given in Figure 5. This time, however, we use several star operators (`<*>`, `*>`, and `<*`) to conveniently compose the semantical results. Recall that `*>` disregards the result of the left parser, `<*` disregards the result

[12] We use different fonts to distinguish the Curry type `BPEL` from the BPEL language.

```
> processS :: Grappa BPMNComp BPEL
> processS = edge (BPMNStart,[n1]) *>
>            flowS (n1,n2) <*
>            edge (BPMNEnd,[n2])
>            where n1,n2 free

> flowS :: (Node,Node)  →  Grappa BPMNComp BPEL
> flowS (n1,n2) = (:[]) <$> flElemS (n1,n2)
> flowS (n1,n2) = (:)   <$> flElemS (n1,n) <*> flowS (n,n2)
>                 where n free

> flElemS :: (Node,Node)  →  Grappa BPMNComp BPELComp
> flElemS (n1,n2) = translateInter itype name <$
>                   edge (BPMNInter itype name, [n1,n2])
>                   where itype,name free
>                         translateInter BPMNWait    = Wait
>                         translateInter BPMNReceive = Receive
> flElemS (n1,n2) = Invoke lab <$
>                   edge (BPMNActivity lab, [n1,n2])
>                   where lab free
> flElemS (n1,n2) = edge (BPMNPGW, [n1,n1t,n1r,n1b]) *>
>                   (Flow <$>
>                    flowS (n1t,n2t) <*>
>                    flowS (n1b,n2b)) <*
>                   edge (BPMNPGW, [n2l,n2t,n2,n2b])
>                   where n1t,n1r,n1b,n2l,n2t,n2b free
> flElemS (n1,n2) = edge (BPMNXGW c1 c2, [n1,n1t,n1r,n1b]) *>
>                   (Switch c1 c2 <$>
>                    flowS (n1t,n2t) <*>
>                    flowS (n1b,n2b)) <*
>                    edge (BPMNXGW d1 d2, [n2l,n2t,n2,n2b])
>                   where c1,c2,d1,d2,n1t,n1r,n1b,n2l,n2t,n2b free
```

Figure 13: BPMN parser that computes BPEL as a result

of the right parser, and `<*>` considers both results and constructs the overall result by function application. For instance, in the definition of the top-level parser `processS`, only the result of the call to `flowS` is relevant and, hence, returned, i.e., the (useless) results of parsing the edges `BPMNStart` and `BPMNEnd` are ignored using the operators `*>` and `<*`, respectively.

The parser transformers `<$>` and `<$` are used to construct the results. As an example, consider `flowS` that constructs a result of type `BPEL`, a list of `BPELComp` elements (i.e., structured BPEL activities). If `flowS` consists only of a single `flElemS` (first case), a singleton list of this element is returned as a result by applying the operation `(:[])` to the element. Otherwise, the result of the first `flElemS` is added to the front of the list of the following `flowS`'s results: Since "`flElemS (n1,n)`" is a parser of type "`Grappa BPMNComp BPELComp`" and the list constructor "`:`" has the polymorphic type "`a → [a] → [a]`", the combined parser "`(:) <$> flElemS (n1,n)`" is of type

```
Grappa BPMNComp ([BPELComp] → [BPELComp])
```

Thus, if we combine this parser with the parser "`flowS (n,n2)`" by the combinator `<*>`, we obtain the desired parser of type "`Grappa BPMNComp BPEL`".

Actually, this definition of `flowS` is a special case of `chain`. Therefore, the parser `flowS` can also be defined using `chain`, which demonstrates the flexibility of graph parser combinators:

```
flowS :: (Node,Node)  →  Grappa BPMNComp BPEL
flowS = chain flElemS
```

Note that the result type of the parser `flElemS` is `BPELComp`. Hence, the parser transformer `<$>` is used to compose an expression of type `BPELComp` from the two sub-flows via the `Flow` or `Switch` constructor, respectively. It should be mentioned that data constructors are also functions of their corresponding types. For instance, `Flow` has type `BPEL → BPEL → BPELComp` so that the expression

```
Flow <$> flowS (n1t,n2t) <*> flowS (n1b,n2b)
```

is well-typed: "`flowS (n1t,n2t)`" is of type "`Grappa BPMNComp BPEL`", hence, "`Flow <$> flowS (n1t,n2t)`" is of type

```
Grappa BPMNComp (BPEL → BPELComp)
```

and, thus, can readily be combined via function application with the subsequent call to `flowS`. Note that the definition of the parser `flElemS` exploits free variables to transfer an attribute value to the result. For instance, consider the second rule defining `flElemS` where a BPEL `Invoke` is constructed from a `BPMNActivity`. Here, the label of the activity is transferred to `Invoke` using the free variable `lab`.

Altogether, one can see from the definitions in Figure 13 that the features of functional logic programming (higher-order functions, free variables, nondeterministic operations) together with an appropriate definition of functional logic graph parser combinators allow for a quite concise and executable specification of a BPMN to BPEL transformation that has been developed step by step from the plain BPMN graph parser of Figure 11.

Since the parser `processS` is not just a specification but also an executable Curry program, we can apply it to the example process of Figure 2 as follows:

```
BPMN2BPEL> processS ex
([Flow [Invoke "act1",Invoke "act2",Receive "ev1"]
       [Invoke "act3",
        Switch "cond1" "cond2" [Invoke "act4",Wait "ev2"]
                               [Invoke "act5"]]], [])
More solutions? [Y(es)/n(o)/a(ll)] y
No more solutions
```

The result is a pair consisting of the semantic BPEL representation of this graph (which has been slightly reformatted to improve its readability) and the remaining graph, which is empty in this case. One can also print the result in XML format, as shown in the definition of the operation `main` (cf. Section 4). The function `bpel2xml` defined in Appendix A actually performs this translation. The indentation performed by `bpel2xml` reflects the tree-structured result of our parser. So, readable and structure-oriented BPEL is generated (cf. Figure 3(b)) and the graph-oriented BPEL features do not need to be used.[13]

7. Discussion and Application

In this section we discuss the applicability of the transformation developed in Section 6. We will describe how the parser shown in Figure 13 can be used for graph completion and how the reverse transformation can be invoked. We also provide results of benchmarks and discuss the scope of this transformation approach. But we have to start with a remark on correctness.

7.1. Identification and Dangling Condition

A problem of the BPMN parsers presented so far is that they accept too many graphs. Further conditions have to be enforced to ensure their correctness [11]:

- Identification condition: Matches have to be injective, i.e., involved nodes have to be pairwise distinct.
- Dangling edge condition: There are no edges in the remaining graph visiting inner nodes of a match.

For instance, the BPMN hypergraphs shown in Figure 14 can also be successfully parsed with the parser given in Figure 11, although they are not members of the language defined by $G_{Process}$ (in the lower example, the second activity would remain after parsing, of course). Even BPMN hypergraphs where all nodes coincide can be parsed, although that way there might be a very large number of possible solutions.

[13]Actually, the graph-oriented BPEL features are not even supported by the definition of the `BPEL` type introduced in this section.

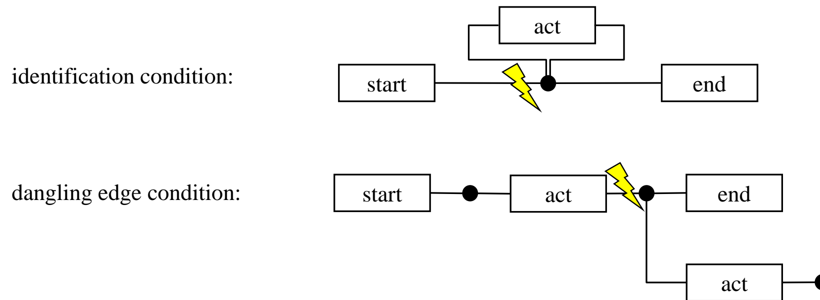


Figure 14: Identification condition and dangling edge condition

In the context of visual languages, it is often convenient to relax the dangling edge condition [47]. This allows for easier specifications. However, from a theoretical point of view, this is not satisfactory. In fact, both conditions can be ensured by additional checks. For instance, to enforce that a particular match is injective, disequality constraints on node variables can be used to ensure that they are pairwise distinct and can never be instantiated to the same node. However, these constraints cannot be globally set but have to be added to the parsers for every single production making them less readable.

Although disequality constraints are very handy, they are not yet part of the language definition of Curry. Currently, only MCC supports them by providing the operator `(=/=)`. However, disequality constraints are an actively discussed topic in the Curry community [4]. With our work on graph parsing, we have identified a practically relevant application that demonstrates the need for disequality constraints as a language extension.[14] For example, a correct definition of the second rule of `flow` (see Figure 11) can be implemented with MCC as follows:

```
flow (n1,n2) | allDifferent [n1,n2,n]  --identification cond
             = flElem (n1,n) *>
               flow (n,n2) *>
               noDanglEdge [n]         --dangling edge cond
               where n free
```

The constraint generator `allDifferent` returns a set of pairwise disequality constraints for the given list of (potentially free) variables. In this case, its result would be {`n1=/=n2, n1=/=n, n2=/=n`}). All these constraints have to be satisfied in order to apply this rule. The operation `noDanglEdge` ensures—also via generating disequality constraints—that the inner nodes of the right-hand side, i.e., the free variables, are not visited by the edges of the remaining graph. With this redefinition, a lot of disequality constraints including many duplicates

[14] The use of the Boolean test operator `(/=)` is not sufficient due to its suspension on free variables, which are typically introduced in the course of the reverse transformation.

are likely to be generated. Thus, it is important that the compiler can deal with them efficiently. Although this is the case for MCC, we do not consider this restriction anymore. It is sufficient to keep in mind that the proposed parser also accepts BPMN graphs where nodes that should be distinct coincide.

7.2. Graph Completion

Our functional logic parsers can perform more tasks beyond graph parsing and semantic computations. As already mentioned, logic-based parsers can be used to perform graph completion. For instance, assume that the edge `("act", [2,6])` in the graph shown in Figure 9 is missing such that this hypergraph is not a member of the language defined by $G_{Process}$ anymore. Then, one can insert a free variable `e` as an edge into the graph and see how `e` is instantiated by the parser. In this example, there are two possible completions that also consume the whole input: `("act", [2,6])` and `("inter", [2,6])`.[15]

```
BPMN2BPEL> (process <* eoi) (e : delete ("act", [2,6]) ex_sm)
            where e free
{e = ("act",[2,6])} ((),[])
More solutions? [Y(es)/n(o)/a(ll)] y
{e = ("inter",[2,6])} ((),[])
More solutions? [Y(es)/n(o)/a(ll)] y
No more solutions
```

Note that this graph completion feature has been put to a good use by connecting it with the DiaGen system where it enables powerful syntax-based user assistance features like diagram completion for DiaGen editors [39, 42].

7.3. Graph Generation

Another useful property of logic-based parsers is their suitability for language generation. Indeed, our graph parsers can also be applied backwards to construct graphs of a given language. For instance, all BPMN graphs up to a particular size can be enumerated:

```
BPMN2BPEL> (process <* eoi) [e1,e2,e3] where e1,e2,e3 free
{e1 = ("start",[_a]), e2 = ("act",[_a,_b]), e3 = ("end",[_b])}
((),[])
More solutions? [Y(es)/n(o)/a(ll)] y
{e1 = ("start",[_c]), e2 = ("end",[_d]), e3 = ("act",[_c,_d])}
((),[])
More solutions? [Y(es)/n(o)/a(ll)] y
{e1 = ("start",[_e]), e2 = ("inter",[_e,_f]), e3 = ("end",[_f])}
((),[])
More solutions? [Y(es)/n(o)/a(ll)] n
```

[15]The operation `delete`, imported from the standard library `List`, deletes the first occurrence of an element in a list.

In these results, free variables are denoted by `_a`, `_b`, etc. This simple example shows one problem when generating graphs. Although there are only two BPMN graphs with three edges (consisting of a start event, an end event, and a single activity or intermediate event, respectively), twelve solutions ($2*3!$) are returned. This undesirable effect is caused by the list representation of graphs. Indeed, the elements of a graph's list representation can be permuted and still describe the very same graph. This problem can be solved by a more sophisticated type for graph parsers, a reformulation of the `edge` primitive, and a slight adaptation of the other primitives. Then a particular number of edges can be pretended while parsing by adding them to a "complement" graph, which is returned as part of the parsing result. This solution has been discussed in [41]. Another solution would be to use a so-called *constraint constructor* [2] for the construction of graphs to enforce that only edge-sorted lists are valid graphs. However, this approach has its weaknesses when dealing with free variables. A quick work-around is the replacement of the `edge` primitive introduced in Section 5.2 by the following variation:

```
edge e (e':g) | e=:=e' = ((), g)
```

Now, the respective edge must always be at the beginning of the list. Although this variant should not be used for parsing (it would fail on all but one representations of a given graph), it is reasonable for language generation and for the reverse transformation. For instance, the previous call would now result in just the two structurally different BPMN graphs:

```
BPMN2BPEL> (process <* eoi) [e1,e2,e3] where e1,e2,e3 free
{e1 = ("start",[_a]), e2 = ("act",[_a,_b]), e3 = ("end",[_b])}
((),[])
More solutions? [Y(es)/n(o)/a(ll)] y
{e1 = ("start",[_c]), e2 = ("inter",[_c,_d]), e3 = ("end",[_d])}
((),[])
More solutions? [Y(es)/n(o)/a(ll)] y
No more solutions
```

7.4. Invoking the Reverse Transformation

Thanks to the logic nature of our parser, it is even possible to map a particular semantic representation back to a graph, e.g., given a term of type `BPEL`, a corresponding BPMN graph can be constructed. In this case, nodes are not instantiated but left as free variables. Actually, the particular node numbers do not matter as long as equal nodes can be identified. Consider the following example calls:

```
BPMN2BPEL> processS [e1,e2,e3,e4] =:= ([Invoke "act1",Wait "ev1"],[])
           where e1,e2,e3,e4 free
{e1 = (BPMNStart,[_a]), e2 = (BPMNActivity "act1",[_a,_b]),
 e3 = (BPMNInter BPMNWait "ev1",[_b,_c]), e4 = (BPMNEnd,[_c])}
More solutions? [Y(es)/n(o)/a(ll)] y
No more solutions
```

```
BPMN2BPEL> processS [e1,e2,e3,e4,e5,e6] =:=
           ([Switch "c1" "c2" [Invoke "act1"] [Invoke "act2"]],[])
           where e1,e2,e3,e4,e5,e6 free
{e1 = (BPMNStart,[_a]), e2 = (BPMNXGW "c1" "c2",[_a,_b,_c,_d]),
 e3 = (BPMNActivity "act1",[_b,_e]), e4 = (BPMNActivity "act2",[_d,_f]),
 e5 = (BPMNXGW _g _h,[_i,_e,_j,_f]), e6 = (BPMNEnd,[_j])}
More solutions? [Y(es)/n(o)/a(ll)] y
No more solutions
```

There are two problems with this approach though: First, the list permutation problem, already discussed in Section 7.3, occurs in this context again. Consequently, the other variation of `edge` has to be used in order to avoid redundant results. Alternatively, the operation `findfirst` could be used so that only the first result is returned (see also the definition of the operation `main` in Section 4). The second issue is that the number of edges needs to be known in advance. This can be avoided by enumerating lists of free variables of increasing length. For instance, a non-deterministic operation that returns an arbitrary list of free variables can be defined as follows:

```
> anyList = []
> anyList = x : anyList where x free
```

Now we can use `anyList` to guess some edge list `es` and pass it to `processS`:

```
BPMN2BPEL> let es free in es =:= anyList &
      processS es =:= ([Switch "c1" "c2" [Invoke "act1",Invoke "act2"]
                                         [Invoke "act3",Wait "ev1"]],[])
{es = [(BPMNStart,[_a]),(BPMNXGW "c1" "c2",[_a,_b,_c,_d]),
 (BPMNActivity "act1",[_b,_e]),(BPMNActivity "act2",[_e,_f]),
 (BPMNActivity "act3",[_d,_g]),(BPMNInter BPMNWait "ev1",[_g,_h]),
 (BPMNXGW _i _j,[_k,_f,_l,_h]),(BPMNEnd,[_l])]}
```

Thanks to the depth-first search strategy, which is the default strategy in most Curry implementations, a graph with a smallest number of edges is computed as a first solution. However, if there is no solution, i.e., if a graph corresponding to the given `BPEL` term does not exist, all graphs of increasing sizes are tried so that the computation does not terminate. This can only be avoided by providing an upper bound on the size of the edge list. However, this is not necessary for our transformation, because for every `BPEL` term there is a corresponding BPMN graph (but not vice versa).

7.5. Performance and Benchmarking

In general, hypergraph parsing has a higher complexity than string parsing, which is known to be less than $\mathcal{O}(n^3)$. Actually, there are even context-free hypergraph languages where parsing is NP-complete [11]. Fortunately, most practically relevant languages are quite efficient to parse. The most efficient existing hypergraph parser we are aware of is part of the DiaGen system [47]. This parser uses dynamic programming techniques. Our functional logic parsers rely on backtracking, which is used in most Curry implementations, and, hence,

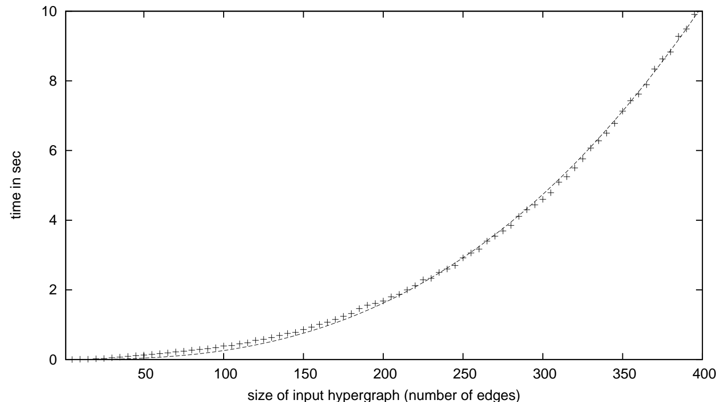


Figure 15: Performance data for parsing activity chains

are not that efficient. Nevertheless, our proposed parsing approach can be applied also to non-toy examples.

Figure 15 shows how graph parser combinators perform for BPMN hypergraphs. We used input processes that contain only a chain of activities between their start and end event, because that way it is possible to grow the input homogeneously. The measurement has been executed on low budget hardware (2GHz Celeron processor, 750MB RAM). As a compiler, MCC has been used. Encapsulated search via the operation `findall` ensures that all possible results are found and, thus, that the order of edges in the list-based graph representation does not matter.

This benchmark shows that a process hypergraph with 200 successive *act* hyperedges created by a generator function can still be parsed in less than two seconds. Moreover, when applied backwards as generators, Grappa parsers are very efficient. Sierpinski triangles have been proposed as a case for performance comparison of graph transformation tools [62]. With a Grappa parser, a regular Sierpinski triangle of generation 12 with $531.441 = 3^{12}$ edges has been generated by MCC in 3 seconds on the same hardware as above. The resulting term contains a large number of free variables. Of course, the parsing of Sierpinski triangles is much more expensive.

Actually, graph parser combinators can even be used as a benchmark for Curry compilers. A precise case description and corresponding performance data of MCC, PAKCS, and KiCS can be found in [41].

7.6. Scope

We have shown by several scenarios the power of our approach in constructing transformations between different modeling languages. Unfortunately, most visual languages are not context-free like structured process models and, thus, cannot be described by a plain hyperedge replacement grammar. Consequently, such languages can only be treated with restrictions or not at all.

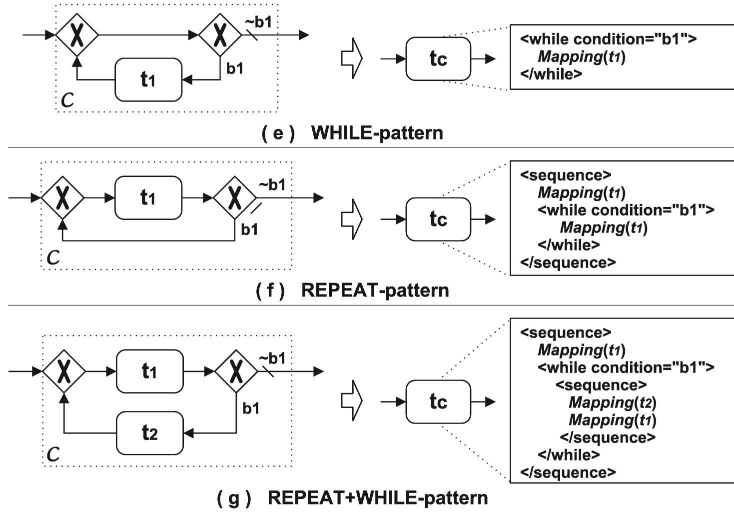


Figure 16: Structured loops as treated in [51]

Regarding business process models, it is easy to add support for further basic activities and structured constructs such as loops, cf. Figure 16. It would also be possible to support gateways with more than two branches. However, non-context-free concepts such as message flow cannot be covered in a straightforward way. Also quasi-structured processes, i.e., processes that can be transformed into structured ones [51], cannot directly be covered. Those need to be preprocessed first. It would be interesting for future work to investigate whether functional logic languages are equally well-suited for the implementation of general purpose graph transformations. Purely functional languages have already been applied in this domain [67] with reasonable success. Since functional logic languages provide even better abstractions for matching patterns in graph structures, we think that they are a natural choice for this purpose.

Although the approach presented so far is limited, business process models are not the only language where it has been put to a good use. We have also applied the Grappa framework to the visual languages of message sequence charts, alligator eggs (a visual notation for lambda calculus [60]), and various tree-like languages such as term graphs without sharing.

8. Related Work

The approach closest to our work is the PETE system, a declarative, purely logic-based approach to transformations recently proposed by Schätz [55]. Similarly to our work, [55] considers a transformation as a relation and presents models as structured terms. The proposed transformation mechanism allows a precise and modular specification of transformations, which can be executed by a Prolog interpreter. An important benefit of this approach is that it is inte-

```
process :- start N1, flow N1 N2, end N2.

flow N1 N2 :- flElem N1 N2.
flow N1 N2 :- flElem N1 N, flow N N2.

flElem N1 N2 :- act N1 N2.
flElem N1 N2 :- inter N1 N2.
flElem N1 N2 :- gw N1 N1t N1r N1b, flow N1t N2t,
                gw N2l N2t N2 N2b, flow N1b N2b.

parseExample :- (start 1, gw 1 2 4 3, act 2 6,
                 act 3 7, gw 5 6 8 7, end 8)
                -o process.
```

Figure 17: BPMN parser in Lolli [27]

grated into the popular Eclipse platform. More precisely, the Eclipse Modeling Framework (EMF) is used as a base, i.e., importers and exporters between EMF models and Prolog terms are provided. However, the resulting Prolog code of a transformation is very verbose and at a rather low level of abstraction. No mapping from a kind of visual transformation rule is provided as in our approach. In order to get the accompanying reverse transformation in PETE, it is required to change the order of predicates on the right-hand side of the rules. The permutation problem caused by the list representation of sets in this article is avoided in PETE by a special set implementation based on ordered lists. PETE allows the transformation of a broader range of models, i.e., it is not restricted to context-free languages. Although there might be many matches of a pattern, backtracking is rarely needed because there are only very few “dead ends”. Most of the time, committing to the first solution is sufficient, because the other solutions would yield the very same result via a different order of matches. PETE has been used to solve the model migration case [56] of this year’s transformation tool contest, i.e., the migration of UML activity diagrams from version 1.4 to version 2.2 [54].

There are also further logic-based approaches to support translations between different structures. An early attempt for the bidirectional translation between abstract and pictorial data has been proposed in [37] where declarative rules written in Prolog have been used to specify the mappings. The idea of representing models by Prolog facts has also been used in [59] for the sake of querying models. In the context of business modeling, Prolog has been used for checking syntactical properties of event-driven process chains [18].

Another interesting related observation is that parsing of visual languages can be modeled and even executed in linear logic [16], a resource-oriented refinement of classical logic. For instance, in [6] the embedding of constraint multiset grammars [36] into linear logic is discussed. However, it seems that hypergraph parsing can be modeled even more straightforwardly. Here, the edges of a hy-

pergraph can be mapped to facts that can be fed into a parser via so-called linear implication ($\multimap$). During the proof, the parser consumes these facts and none of them must be left at the end. Figure 17 shows a BPMN graph parser implemented in the linear logic programming language Lolli [27] (the two different gateway types are not distinguished in this program). The main predicate of this program is `parseExample`, which analyzes the example graph shown in Figure 9. Not even a parser framework is required with this approach. The graph parser combinators described in this article also hide the remaining resources from the user. Their major benefits are their flexible application, the reversibility of the resulting parsers, and the possibility to compute semantic representations in a straightforward way.

Another related approach is architectural design rewriting (ADR) [9]. Here, hypergraphs are used for representing the hierarchical architecture of a system. Hyperedge replacement grammars are used for building configurations in a way that avoids, e.g., cyclic or broken flows. The involved graphs can often be represented as terms. In this case, no parsing is required. Moreover, an algebra on these terms can be defined and exploited afterwards. Graph transformation rules can be used for manipulating configurations. Then, it has to be proven that the rules of the underlying algebra are preserved. The developers of the ADR approach also admit that not everything can be modeled with ADR, but if a part of a problem can be captured, ADR provides an elegant way to reason about and reconfigure the system.

Regarding error correction, the purely functional HUT library [61] needs to be mentioned. It supports the construction of (string) parser combinators with powerful error correction mechanisms. There, a parser does never fail but rather constructs a minimal sequence of correction steps. We have shown how functional logic graph parser combinators provide specific kinds of error handling. Redundant edges, for instance, may just remain at the end. This is already quite powerful, since, in contrast to strings, graphs are sets of components, i.e., no particular order on edges or nodes is imposed. Thus, it does not matter where the redundant components are placed. Furthermore, due to their logic nature, graph parser combinators allow to deal with further errors in a convenient way, e.g., they can complete incomplete graphs. An alternative approach for hypergraph completion has been proposed in [38]. This approach is based on dynamic programming.

Other approaches for the specification of bidirectional transformations include Schürr's triple graph grammars [57] and Pierce's lense combinators [14] that even support the synchronization of evolving models. However, whereas the former requires much more specification effort than our approach, the latter is quite restricted in its applicability.

Finally, the competing solutions of the GraBaTs 2009 synthesis case have to be mentioned as related work, of course. The case was won by a plain Java solution that covered the most comprehensive subset of BPMN [15]. The best general transformation approaches have been GROOVE, MoTMoT, and Grappa. The evaluation matrix can be found at `http://is.tm.tue.nl/staff/pvgorp/events/grabats2009/synthesis-evaluation.pdf` (accessed on 2010-07-30).

9. Conclusion

In this article, we have proposed a novel approach to construct a bidirectional transformation between BPMN and BPEL and discussed its potential applicability. Both languages and the transformation in-between are highly relevant. Our solution is based on functional logic programming techniques and implemented using the multi-paradigm declarative language Curry. The proposed transformation relies on an extended version of the graph parser combinator framework Grappa, which was originally introduced in [40]. Grappa requires diagrams—process models in this case—to be represented by hypergraphs. The article has shown that hyperedge replacement grammars, a context-free device for the specification of graph languages, can be translated into graph parsers in a systematic way. The resulting parsers closely resemble the grammar rules. A parser for a significant subset of the BPMN language (structured process models) has been realized that way.

Moreover, this case study has shown the flexibility of graph parser combinators in the construction of semantic representations for the first time. To this end, Grappa has been extended by typed hypergraphs. With the resulting transformation, BPMN graphs cannot only be derived backwards from a given BPEL structure, but it is also possible to complete partial process models or to use the parser for language generation. Since the whole Grappa framework are only a few dozen lines of code, this article contains the complete code for running the transformations. Actually, the article's source text is a literate Curry program and, as such, directly executable.

Caballero and López-Fraguas [10] have recognized the benefits of functional logic languages for building *string* parsers. However, in the context of graphs they are even more useful. Graph parser combinators make heavy use of key features of both the functional and the logic programming approach: Higher-order functions allow the treatment of parsers as first class citizens, and non-determinism and free variables are beneficial for dealing with errors and incomplete information. Therefore, graph parser combinators are also an interesting and promising application area of functional logic programming and, at the same time, an example how logic programming techniques advance the state-of-the-art in visual languages.

For future work, it would be useful to investigate how functional logic languages can be exploited to implement more general graph transformation systems to overcome the limitations in scope of the approach developed so far. Moreover, the transformation proposed in this article constructs the term representations of models from scratch again and again. It would also be interesting to see how functional logic techniques could be used in order to support the reuse of existing structures, i.e., allow for model synchronization.

To make this research reproducible, an installation of the presented solution is available as a SHARE virtual machine (Sharing Hosted Autonomous Research Environments). SHARE images can be accessed via `http://is.tm.tue.nl/staff/pvgorp/share/` (accessed on 2010-07-30). The machine is called `Ubuntu-8.10_GB9_grappa-bpm.vdi`.

[1] Antoy, S., Echahed, R., Hanus, M., 2000. A needed narrowing strategy. Journal of the ACM 47 (4), 776–822.

[2] Antoy, S., Hanus, M., 2002. Functional logic design patterns. In: Proc. of the 6th International Symposium on Functional and Logic Programming. Vol. 2441 of LNCS. Springer-Verlag, pp. 67–87.

[3] Antoy, S., Hanus, M., 2010. Functional logic programming. Communications of the ACM 53 (4), 74–85.

[4] Arias, E. J. G., Carballo, J. M., Poza, J. M. R., 2007. A proposal for disequality constraints in Curry. Electronic Notes in Theoretical Computer Science 177, 269–285, proc. of the 15th International Workshop on Functional and (Constraint) Logic Programming.

[5] Bardohl, R., Minas, M., Taentzer, G., Schürr, A., 1999. Application of graph transformation to visual languages. In: Handbook of Graph Grammars and Computing by Graph Transformation. Vol. 2: Applications, Languages, and Tools. World Scientific, pp. 105–180.

[6] Bottoni, P., Meyer, B., Marriott, K., Parisi-Presicce, F., 2001. Deductive parsing of visual languages. In: Proc. of the 4th International Conference on Logical Aspects of Computational Linguistics. Vol. 2099 of LNCS. Springer-Verlag, pp. 79–94.

[7] Braßel, B., Hanus, M., Müller, M., 2008. High-level database programming in Curry. In: Proc. of the Tenth International Symposium on Practical Aspects of Declarative Languages. Vol. 4902 of LNCS. Springer-Verlag, pp. 316–332.

[8] Braßel, B., Huch, F., 2009. The Kiel Curry system KiCS. In: Applications of Declarative Programming and Knowledge Management. Vol. 5437 of LNCS. Springer-Verlag, pp. 195–205.

[9] Bruni, R., Lafuente, A. L., Montanari, U., 2009. Hierarchical design rewriting with Maude. Electronic Notes in Theoretical Computer Science 238 (3), 45 – 62, proceedings of the Seventh International Workshop on Rewriting Logic and its Applications.

[10] Caballero, R., López-Fraguas, F. J., 1999. A functional-logic perspective of parsing. In: Proc. of the 4th Fuji International Symposium on Functional and Logic Programming. Vol. 1722 of LNCS. Springer-Verlag, pp. 85–99.

[11] Drewes, F., Habel, A., Kreowski, H.-J., 1997. Hyperedge replacement graph grammars. In: Handbook of Graph Grammars and Computing by Graph Transformation. Vol. I: Foundations. World Scientific, Ch. 2, pp. 95–162.

[12] Dumas, M., 2009. Case study: BPMN to BPEL model transformation. `http://is.ieis.tue.nl/staff/pvgorp/events/grabats2009/cases/grabats2009synthesis.pdf` (accessed on 2010-07-11).

[13] Fischer, S., 2005. A functional logic database library. In: Proc. of the ACM SIGPLAN 2005 Workshop on Curry and Functional Logic Programming. ACM Press, pp. 54–59.

[14] Foster, J. N., Greenwald, M. B., Moore, J. T., Pierce, B. C., Schmitt, A., 2005. Combinators for bi-directional tree transformations: A linguistic approach to the view update problem. In: Proc. of the 32nd ACM Symposium on Principles of Programming Languages. ACM, pp. 233–246.

[15] García-Bañuelos, L., 2009. Translating BPMN models to BPEL code. Accepted as a solution for the synthesis case of the GraBaTs 2009 tool contest.

[16] Girard, J.-Y., 1987. Linear logic. Theoretical Computer Science 50, 1–102.

[17] González-Moreno, J., Hortalá-González, M., López-Fraguas, F., Rodríguez-Artalejo, M., 1999. An approach to declarative programming based on a rewriting logic. Journal of Logic Programming 40, 47–87.

[18] Gruhn, V., Laue, R., 2006. Validierung syntaktischer und anderer EPK-Eigenschaften mit PROLOG. In: Proc. des 5. GI Workshops über Geschäftsprozessmanagement mit Ereignisgesteuerten Prozessketten. Vol. 224 of CEUR Workshop Proceedings. CEUR-WS.org, pp. 69—85.

[19] Gruhn, V., Laue, R., 2007. What business process modelers can learn from programmers. Science of Computer Programming 65 (1), 4–13.

[20] Hanus, M., 2000. A functional logic programming approach to graphical user interfaces. In: Proc. of the Second International Workshop on Practical Aspects of Declarative Languages. Vol. 1753 of LNCS. Springer-Verlag, pp. 47–62.

[21] Hanus, M., 2001. High-level server side web scripting in Curry. In: Proc. of the Third International Symposium on Practical Aspects of Declarative Languages. Vol. 1990 of LNCS. Springer-Verlag, pp. 76–92.

[22] Hanus, M., 2006. Curry: An Integrated Functional Logic Language (Version 0.8.2). `http://www.curry-language.org/` (accessed on 2010-07-11).

[23] Hanus, M., 2006. Type-oriented construction of web user interfaces. In: Proc. of the 8th ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming. ACM Press, pp. 27–38.

[24] Hanus, M., 2007. Multi-paradigm declarative languages. In: Proc. of the 23rd International Conference on Logic Programming. Vol. 4670 of LNCS. Springer-Verlag, pp. 45–75.

[25] Hanus, M., Antoy, S., Braßel, B., Engelke, M., Höppner, K., Koj, J., Niederau, P., Sadre, R., Steiner, F., 2010. PAKCS: The Portland Aachen Kiel Curry System. Available at `http://www.informatik.uni-kiel.de/~pakcs/` (accessed on 2010-07-11).

[26] Hanus, M., Koschnicke, S., 2010. An ER-based framework for declarative web programming. In: Proc. of the 12th International Symposium on Practical Aspects of Declarative Languages. Vol. 5937 of LNCS. Springer-Verlag, pp. 201–216.

[27] Hodas, J. S., Miller, D., 1994. Logic programming in a fragment of intuitionistic linear logic. Information and Computation 110 (2), 327–365.

[28] Hutton, G., 1992. Higher-order functions for parsing. Journal of Functional Programming 2 (3), 323–343.

[29] Johnson, R., Pearson, D., Pingali, K., 1994. The program structure tree: Computing control regions in linear time. In: Proc. of the ACM SIGPLAN Conference on Programming Language Design and Implementation. ACM Press, pp. 171–185.

[30] Johnson, S. C., 1975. Yacc: Yet another compiler compiler. Tech. Rep. 32, Bell Laboratories, Murray Hill, New Jersey.

[31] Kasami, T., 1965. An efficient recognition and syntax-analysis algorithm for context free languages. Scientific Report AF CRL-65-758, Air Force Cambridge Research Laboratory.

[32] Knuth, D. E., 1968. Semantics of context-free languages. Theory of Computing Systems 2 (2), 127–145.

[33] Knuth, D. E., 1984. Literate programming. The Computer Journal 27 (2), 97–111.

[34] Leijen, D., Meijer, E., 2001. Parsec: Direct style monadic parser combinators for the real world. Tech. Rep. UU-CS-2001-27, Department of Computer Science, Universiteit Utrecht.

[35] Lux, W., 1999. Implementing encapsulated search for a lazy functional logic language. In: Proc. of the 4th Fuji International Symposium on Functional and Logic Programming. Vol. 1722 of LNCS. Springer-Verlag, pp. 100–113.

[36] Marriott, K., 1994. Constraint multiset grammars. In: Proc. of the 1994 IEEE Symposium on Visual Languages. IEEE, pp. 118–125.

[37] Matsuoka, S., Takahashi, S., Kamada, T., Yonezawa, A., 1992. A general framework for bidirectional translation between abstract and pictorial data. ACM Transactions on Information Systems 10 (4), 408–437.

[38] Mazanek, S., Maier, S., Minas, M., 2008. An algorithm for hypergraph completion according to hyperedge replacement grammars. In: Proc. of the 4th International Conference on Graph Transformations. Vol. 5214 of LNCS. Springer-Verlag, pp. 39–53.

[39] Mazanek, S., Maier, S., Minas, M., 2008. Auto-completion for diagram editors based on graph grammars. In: Proc. of the 2008 IEEE Symposium on Visual Languages and Human-Centric Computing. IEEE, pp. 242–245.

[40] Mazanek, S., Minas, M., 2008. Functional-logic graph parser combinators. In: Proc. of the 19th International Conference on Rewriting Techniques and Applications. Vol. 5117 of LNCS. Springer-Verlag, pp. 261–275.

[41] Mazanek, S., Minas, M., 2008. Graph parser combinators: A challenge for Curry-compilers. In: Hanus, M., Fischer, S. (Eds.), 25. Workshop der GI-Fachgruppe "Programmiersprachen und Rechenkonzepte". Christian-Albrechts-Universität zu Kiel, pp. 55–66, Tech. Rep. 0811.

[42] Mazanek, S., Minas, M., 2009. Business process models as a showcase for syntax-based assistance in diagram editors. In: Proc. of the 12th International Conference on Model Driven Engineering Languages and Systems. Vol. 5795 of LNCS. Springer-Verlag, pp. 322–336.

[43] Mazanek, S., Minas, M., 2009. Transforming BPMN to BPEL using parsing and attribute evaluation with respect to a hypergraph grammar. Accepted as a solution for the synthesis case of the GraBaTs 2009 tool contest.

[44] Mendling, J., Reijers, H., van der Aalst, W., 2009. Seven process modeling guidelines (7pmg). Information and Software Technology 52 (2), 127–136.

[45] Minas, M., 2000. Creating semantic representations of diagrams. In: Proc. of the International Workshop on Applications of Graph Transformations with Industrial Relevance. Vol. 1779. Springer-Verlag, pp. 209–224.

[46] Minas, M., 2000. Hypergraphs as a uniform diagram representation model. In: Proc. of the 6th International Workshop on Theory and Application of Graph Transformations. Vol. 1764 of LNCS. Springer-Verlag, pp. 281–295.

[47] Minas, M., 2002. Concepts and realization of a diagram editor generator based on hypergraph transformation. Science of Computer Programming 44 (2), 157–180.

[48] OASIS, 2007. Web Services Business Process Execution Language Version 2.0. `http://docs.oasis-open.org/wsbpel/2.0/wsbpel-v2.0.pdf` (accessed on 2010-07-11). Actually, the transformation of this article does not generate BPEL 2.0 but still BPEL 1.1.

[49] Object Management Group, 2009. Business Process Modeling Notation (BPMN). `http://www.omg.org/spec/BPMN/1.2/` (accessed on 2010-07-11).

[50] Ouyang, C., Dumas, M., Breutel, S., ter Hofstede, A., 2006. Translating standard process models to BPEL. In: Proc. of the 18th International Conference on Advanced Information Systems Engineering. Vol. 4001 of LNCS. Springer-Verlag, pp. 417–432.

[51] Ouyang, C., Dumas, M., ter Hofstede, A., van der Aalst, W., 2007. Pattern-based translation of BPMN process models to BPEL web services. International Journal of Web Services Research 5 (1), 42–62.

[52] Peyton Jones, S., 2003. Haskell 98, Language and Libraries, The Revised Report. Cambridge University Press.

[53] Pratt, T. W., 1971. Pair grammars, graph languages and string-to-graph translations. Journal of Computer and System Sciences 5 (6), 560–595.

[54] Rose, L. M., Kolovos, D. S., Paige, R. F., Polack, F. A., 2010. Model migration case for TTC 2010. `http://is.ieis.tue.nl/staff/pvgorp/events/TTC2010/cases/ttc2010_submission_2_v2010-04-22.pdf` (accessed on 2010-07-11).

[55] Schätz, B., 2009. Formalization and rule-based transformation of EMF Ecore-based models. In: Proc. of the First International Conference on Software Language Engineering. Vol. 5452 of LNCS. Springer-Verlag, pp. 227–244.

[56] Schätz, B., 2010. UML model migration with PETE. `http://is.ieis.tue.nl/staff/pvgorp/events/TTC2010/submissions/final/pete.pdf` (accessed on 2010-07-11).

[57] Schürr, A., 1994. Specification of graph translators with triple graph grammars. In: Proc. of the 20th International Workshop on Graph-Theoretic Concepts in Computer Science. Vol. 903 of LNCS. Springer-Verlag, pp. 151–163.

[58] Slagle, J., 1974. Automated theorem-proving for theories with simplifiers, commutativity, and associativity. Journal of the ACM 21 (4), 622–642.

[59] Störrle, H., 2007. A PROLOG-based approach to representing and querying software engineering models. In: Proc. of the VLL 2007 Workshop on Visual Languages and Logic. Vol. 274 of CEUR Workshop Proceedings. CEUR-WS.org, pp. 71–83.

[60] Strobl, T., Minas, M., 2009. Implementing an animated lambda-calculus. In: Proc. of the Workshop on Visual Languages and Logic. Vol. 510 of CEUR Workshop Proceedings. CEUR-WS.org.

[61] Swierstra, S. D., Azero Alcocer, P. R., 1999. Fast, error correcting parser combinators: a short tutorial. In: Proc. of the 26th Seminar on Current Trends in Theory and Practice of Informatics. Vol. 1725 of LNCS. Springer-Verlag, pp. 111–129.

[62] Taentzer, G., Biermann, E., Bisztray, D., Bohnet, B., Boneva, I., Boronat, A., Geiger, L., Geiß, R., Horvath, Á., Kniemeyer, O., Mens, T., Ness, B., Plump, D., Vajk, T., 2008. Generation of Sierpinski triangles: A case study

for graph transformation tools. In: Proc. of the Third International Symposium on Applications of Graph Transformations with Industrial Relevance. Vol. 5088 of LNCS. Springer-Verlag, pp. 514–539.

[63] Tanaka, T., 1991. Definite-clause set grammars: a formalism for problem solving. Journal of Logic Programming 10 (1), 1–17.

[64] Vanhatalo, J., Völzer, H., Koehler, J., 2008. The refined process structure tree. In: Proc. of the 6th International Conference on Business Process Management. Vol. 5240 of LNCS. Springer-Verlag, pp. 100–115.

[65] Wadler, P., 1985. How to replace failure by a list of successes: A method for exception handling, backtracking, and pattern matching in lazy functional languages. In: Functional Programming Languages and Computer Architecture. Vol. 201 of LNCS. Springer-Verlag, pp. 113–128.

[66] Wallace, M., 2008. Partial parsing: Combining choice with commitment. In: Proc. of the 19th International Workshop on the Implementation and Application of Functional Languages. Vol. 5083 of LNCS. Springer-Verlag, pp. 93–110.

[67] West, S., Kahl, W., 2009. A generic graph transformation, visualisation, and editing framework in Haskell. In: Proc. of the 8th International Workshop on Graph Transformation and Visual Modeling Techniques. Vol. 18 of Electronic Communications of the EASST. European Association of Software Science and Technology.

Appendix A. Generating BPEL-XML from Constructor Terms

For the sake of completeness, we provide an implementation of the generation of BPEL-XML from `BPEL` terms as defined in this article. There are also higher-level approaches for the generation of XML in functional languages but this simple solution is sufficient for our purposes.

```
> bpel2xml :: BPEL  →  String
> bpel2xml f = "<process>\n" ++
>                  indent 1 (seq2xml f) ++
>              "</process>\n"

> seq2xml [e] = bpelComp2xml e
> seq2xml es@(_:_:_) = "<sequence>\n" ++
>                          indent 1 (concatMap bpelComp2xml es) ++
>                      "</sequence>\n"

> bpelComp2xml (Invoke name) = "<invoke name=\""++name++"\"/>\n"
> bpelComp2xml (Wait name) = "<wait name=\""++name++"\"/>\n"
> bpelComp2xml (Receive name) = "<receive name=\""++name++"\"/>\n"
> bpelComp2xml (Flow f1 f2) = "<flow>\n" ++
>                                indent 1 (seq2xml f1) ++
>                                indent 1 (seq2xml f2) ++
>                              "</flow>\n"
> bpelComp2xml (Switch c1 c2 f1 f2) =
>                              "<switch>\n" ++
>                              " <case cond=\""++c1++"\">\n" ++
>                                indent 1 (seq2xml f1) ++
>                              " </case>\n" ++
>                              " <case cond=\""++c2++"\">\n" ++
>                                indent 1 (seq2xml f2) ++
>                              " </case>\n" ++
>                              "</switch>\n"

> --lines and unlines are inverse functions for switching between a
> --text with line breaks '\n' and the respective list of lines
> indent n s = unlines (map (nblanks++) (lines s))
>              --construct a string of n blanks
>              where nblanks = take n (repeat ' ')
```