

Constructing a Bidirectional Transformation between BPMN and BPEL with Functional-logic Graph Parser Combinators

Steffen Mazanek and Mark Minas

Universität der Bundeswehr München, Germany,
{`steffen.mazanek|mark.minas`}@unibw.de

Abstract. The abstract syntax of structured Business Process Models (BPMs) can be described by a context-free hypergraph grammar in a straightforward way. Functional-logic graph parser combinators can be used to construct powerful parsers for such context-free hypergraph grammars. These parsers can be enriched with semantic computations, e.g. to synthesize BPEL from BPMN. Moreover, they are bidirectional by default thanks to their logic nature. Finally, functional-logic parsers stand out by their direct support for completion and language generation.

1 Introduction

This paper describes how BPMs [1] can be translated to BPEL [2] using a framework of functional-logic graph parser combinators [3]. The proposed approach requires the BPMs to be structured, i.e. composed only by sequential and parallel composition of activities as well as structured loops. Such BPMs can be described with a context-free hypergraph grammar, which in turn can be straightforwardly translated into a parser using functional-logic graph parser combinators.

In the following, BPMs are modeled as hypergraphs [4], so that hypergraph parsing can be used for syntax analysis. The provided implementation has been realized using the multi-paradigm programming language Curry [5].

2 Hypergraphs as a Model for BPMs

Fig. 1 shows an example process, which is well-structured. It contains several activities, but also different kinds of intermediate events (namely wait and receive). Furthermore, it comprises both a parallel and an exclusive branching.

Hypergraphs are known to be well-suited as a diagram representation model [4]. As an example consider the BPM hypergraph shown in Fig. 2, which corresponds to Fig. 1. Hyperedges are represented as boxes and nodes as (numbered) black dots. If a node is attached to an edge, it is connected to this edge by a line, which is also called tentacle in this context. Tentacles are numbered to indicate the roles of the connected nodes. For instance, activities (edge label *act*) always

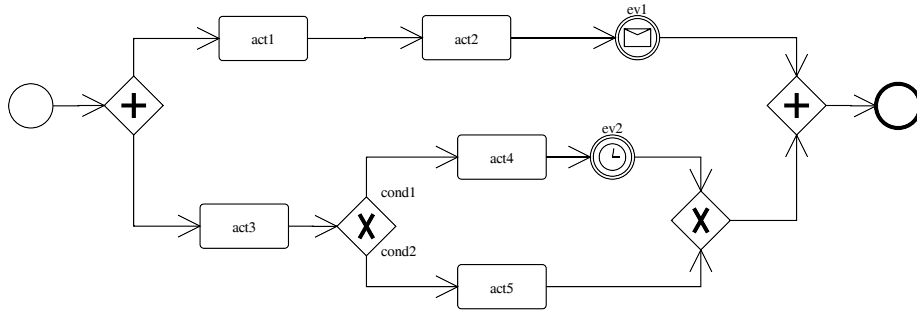


Fig. 1. Example process

have an incoming (1) and an outgoing tentacle (2); parallel and exclusive gateways (edge label *pgw*/*xgw*) always have to have four tentacles, which basically correspond to the corners of the concrete diamond-shaped diagram component (left (1), top (2), right (3), and bottom (4)).

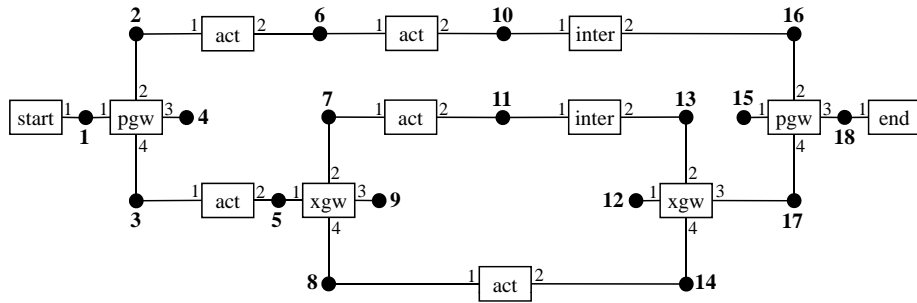


Fig. 2. BPM Hypergraph of Fig. 1

The hypergraph language of well-structured BPMs can be defined using a hyperedge replacement grammar (HRG). Each HRG consists of two finite sets of terminal and nonterminal hyperedge labels and a starting hypergraph, which contains only a single nonterminally labeled hyperedge. Syntax is described by a set of hypergraph productions. Fig. 3 shows the productions of the HRG $G_{Process}$. For the sake of conciseness, loops are not considered. Productions $L ::= R_1, L ::= R_2, \dots$ with the same left-hand side are drawn as $L ::= R_1 | R_2 | \dots$. 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. This grammar can be mapped to a functional-logic parser in a straightforward way as will be shown in the next section.

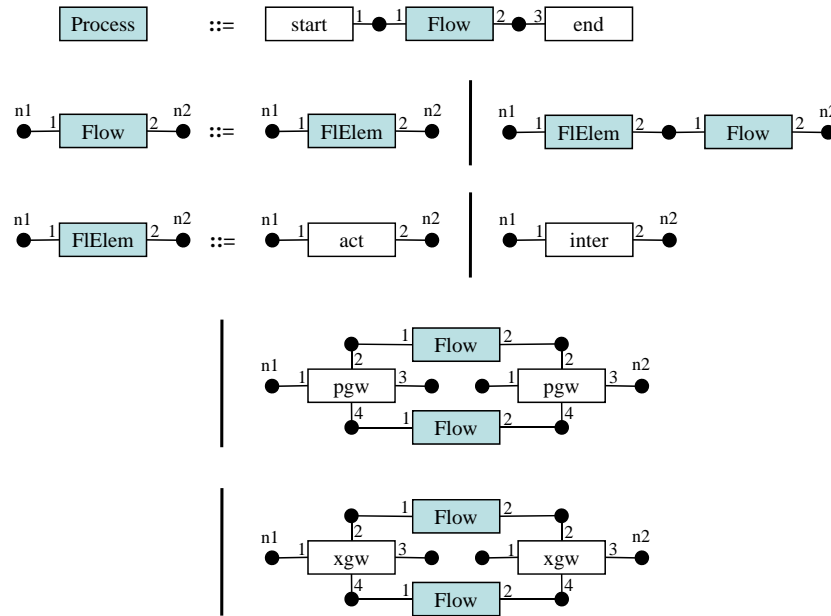


Fig. 3. Productions of $G_{Process}$

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, cf. [6]. Matching node labels of left-hand side and right-hand side determine how the right-hand side has to fit in after removing e . The hypergraph language generated by a grammar is defined by the set of terminally labeled hypergraphs that can be derived from the starting hypergraph. For $G_{Process}$ this is just the set of all BPM hypergraphs.

3 Graph Parser Combinators revisited

The framework graph parser combinators¹ has been realized in the functional logic programming language Curry [5]. Curry is a declarative multi-paradigm language combining interesting features from both functional and logic programming [7]. The Curry syntax is very close to Haskell. The main addition are free (logic) variables in conditions and right-hand sides of defining rules. A Curry program consists of definitions of functions and data types on which these functions operate. Functions are defined by conditional equations with constraints in the conditions. They are evaluated lazily and can be called with partially instantiated arguments. Function calls with free variables are evaluated by a possibly non-deterministic instantiation of the required arguments, i.e. arguments whose

¹ <http://www.unibw.de/steffen.mazanek/forschung/grappa>

```

data BPMInterKind = BPMWait | BPMReceive
data BPMComp = BPMStart | BPMEnd |
              BPMPGW | BPMXGW String String |
              BPMActivity String |
              BPMInter BPMInterKind String
type BPMGraph = Graph BPMComp

```

Fig. 4. Data declarations required for the description of BPM graphs

values are necessary to decide the applicability of a rule. This mechanism is called *narrowing* [8].

The complete code of the combinator framework used in the following is given in App. A. How it works is described in depth in [3]. Basically, graphs are represented as edge lists where each edge is accompanied by a list of attachment nodes. The tentacle number thereby corresponds to the position of a node in this list. In the simplest case, edges can be represented just by strings corresponding to their labels. However, for the transformation of BPMs into BPEL edges have to carry attributes, e.g. the label of an activity or the kind of an intermediate event. Therefore, BPM edges are represented by a newly introduced type `BPMComp`, cf. Fig. 4. Actually, the framework given in App. A adds support for typed edges to the original version presented in [3]. The representation of the example graph of Fig. 2 is given below (including an optional type declaration):

```

ex :: BPMGraph
ex = [(BPMStart, [1]), (BPMPGW, [1,2,4,3]), (BPMActivity "act1", [2,6]),
      (BPMActivity "act2", [6,10]), (BPMInter BPMReceive "ev1", [10,16]),
      (BPMActivity "act3", [3,5]), (BPMXGW "cond1" "cond2", [5,7,9,8]),
      (BPMActivity "act4", [7,11]), (BPMActivity "act5", [8,14]),
      (BPMInter BPMWait "ev2", [11,13]), (BPMXGW "" "", [12,13,17,14]),
      (BPMPGW, [15,16,18,17]), (BPMEnd, [18])]

```

Note that the actual order of edges in a graph's list representation does not matter. So, there are a lot of terms describing the very same graph. This approach is easiest to implement and understand, but also has some weaknesses. Those will be discussed in Sect. 5.

The basic principle of graph parser combinators is the same as in the string setting. Primitive parsers are provided that can be combined into more advanced parsers using a set of powerful combinators. For example, string parser combinator libraries usually provide a sequence and a choice combinator. Therewith, parsers can be constructed that closely resemble the respective grammars.

Parser combinators are very popular, because they integrate seamlessly with the rest of the program and hence the full power of the host language can be used. Unlike generators such as Yacc no extra formalism is needed to specify the grammar. Functional languages are particularly well-suited for the implementation of combinator libraries. Here, a parser is just a function. A combinator

like choice then is a higher-order function, i.e., a function whose parameters are functions again. In the context of graph parsing, however, also concepts from logic programming languages are required as discussed in [3].

4 Solution

Since the required BPM parser has to transform BPM into BPEL, the target data structure first has to be modeled. A significant subset of BPEL can be represented as follows:

```
type BPEL = [BPELComp]
data BPELComp = Invoke String |
               Wait String | Receive String |
               Flow BPEL BPEL |
               Switch String String BPEL BPEL
```

Whereas the source data structure is graph-based (`Graph BPMComp`), the target structure basically is a tree and, thus, can be recursively defined as shown above. Note that terms of type `BPEL` can be transformed to BPEL-XML in a straightforward way as demonstrated in App. B. One could also directly construct XML while parsing. However, the language Curry allows for more flexibility when dealing with constructor terms like that.

Fig. 5 shows the developed parser for BPM graphs. As already mentioned it resembles the grammar given in Fig. 3 very closely. The primitive `edge` is used to “consume” a single edge. Nonterminals are mapped to recursive function calls, e.g. `flow`. Choice does not need to be represented by an extra combinator. Rather it is sufficient to declare several function bodies. The star operators `<*>`, `*>` and `<*` represent the sequential composition of parsers (which is not really sequential in the context of graphs though). The reason for the existence of three different star operators is only that results can be composed more conveniently. So `*>` disregards the result of the left parser, `<*` disregards the result of the right parser and `<*>` considers both results. So, in the definition of the top-level parser `process` only the result of the call to `flow` is relevant (and passed through).

Note also how nodes are treated. Whereas nodes of the left-hand side of a production are mapped to function parameters, the inner nodes of the right-hand side are declared as free variables. Those are instantiated while parsing. Here, actually comes logic programming into play for the first time.

Finally, the used combinators `<$>` and `<$` are used to construct results. Whereas `<$>` applies a function to the result of the following parser, `<$` just returns a constant value. As an example consider `flow` that constructs a result of type `BPEL`, which is a list of `BPELComps`. If the `flow` consists of just a single `f1Elem` (first case), as a result a singleton list of this very element is returned. Otherwise, the result of the first `f1Elem` is added to the front of the list of the following `flow`’s results (roughly speaking).

Also note how free variables can be used to transfer an attribute value into the result. This can be best seen in the case of `f1Elem` where an `Invoke` is

```

process :: Grappa BPMComp BPEL
process = edge (BPMStart, [n1]) *>
  flow (n1,n2) <*>
  edge (BPMEnd, [n2])
  where n1,n2 free

flow :: (Node,Node) -> Grappa BPMComp BPEL
flow (n1,n2) = (:[]) <$> flElem (n1,n2)
flow (n1,n2) = (:) <$>
  flElem (n1,n) <*>
  flow (n,n2)
  where n free

inter BPMWait = Wait
inter BPMReceive = Receive
flElem :: (Node,Node) -> Grappa BPMComp BPELComp
flElem (n1,n2) = inter itype name
  <$ edge (BPMInter itype name, [n1,n2])
  where itype,name free
flElem (n1,n2) = Invoke lab
  <$ edge (BPMActivity lab, [n1,n2])
  where lab free

flElem (n1,n2) =
  edge (BPMPGW, [n1,n1t,n1r,n1b]) *>
  edge (BPMPGW, [n2l,n2t,n2,n2b]) *>
  (Flow <$>
    flow (n1t,n2t) <*>
    flow (n1b,n2b))
  where n1t,n1r,n1b,n2l,n2t,n2b free
flElem (n1,n2) =
  edge (BPMXGW c1 c2, [n1,n1t,n1r,n1b]) *>
  edge (BPMXGW c1_ c2_, [n2l,n2t,n2,n2b]) *>
  (Switch c1 c2 <$>
    flow (n1t,n2t) <*>
    flow (n1b,n2b))
  where c1,c2,c1_,c2_,n1t,n1r,n1b,n2l,n2t,n2b free

```

Fig. 5. BPM parser constructed with graph parser combinators

constructed. Here, the label `lab` of the corresponding `BPMActivity` is transferred to `Invoke` using a free variable.

The parser `process` can be applied to the example hypergraph of Fig. 2 as follows (the pretty printing of the result has been done manually and not by the Curry interpreter):

```
BPM> process 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)] ;
No more solutions
```

5 Discussion

Bidirectionality: Thanks to the logic nature of the proposed parser some bidirectionality is gained for free. Consider the following example calls:

```
BPM> process [e1,e2,e3,e4]=:([Invoke "act1",Wait "ev1"],[])
      where e1,e2,e3,e4 free
{e1 = (BPMStart,[_a]), e2 = (BPMActivity "act1",[_a,_b]),
  e3 = (BPMInter BPMWait "ev1",[_b,_c]), e4 = (BPMEnd,[_c])}
More solutions? [Y(es)/n(o)/a(ll)] n

BPM> process [e1,e2,e3,e4,e5,e6]=:
      ([Switch "c1" "c2" [Invoke "act1"] [Invoke "act2"]],[])
      where e1,e2,e3,e4,e5,e6 free
{e1 = (BPMStart,[_a]), e2 = (BPMXGW "c1" "c2",[_a,_b,_c,_d]),
  e3 = (BPMXGW _e _f,[_g,_h,_i,_j]), e4 = (BPMActivity "act1",[_b,_h]),
  e5 = (BPMActivity "act2",[_d,_j]), e6 = (BPMEnd,[_i])}
More solutions? [Y(es)/n(o)/a(ll)] n
```

In the results, free variables are denoted by `_a`, `_b`, etc. There are two major problems with this approach though: First, the number of edges needs to be known in advance. And second, since graphs are represented as lists, all permutations of this solution are also returned (here 24). Whereas the second problem can be solved by just returning the first solution at a time (`head . findall`), there is no straightforward solution to the second problem. Maybe control of search as provided by most Curry interpreters can be exploited here, i.e. the use of a breadth-first strategy instead of depth-first search for solutions. However, this has not been investigated yet.

Completion and Language Generation: A special feature of the presented BPM parser is that it provides support for completion and language generation out

of the box (actually, for language generation a more sophisticated framework of graph parser combinators has been introduced in [9], which does not suffer from the list permutation problem). Consider the following call as an example:

```
BPM> process (e:tail ex) where e free
{e = (BPMStart, [1])}
([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)] ;
No more solutions
```

The first edge of the example hypergraph is removed using `tail` and a free variable edge `e` is introduced instead. As a result, `e` is instantiated as just the missing edge. Any other missing edge would have been derived equally well. Since this paper is about transformations, completion is not further discussed though. More information about this can be found in [3]. Note that in [10] this benefit of functional-logic parsers has already been recognized for string languages. In the context of graphs it is even more useful though.

Performance: Hypergraph parsing generally has a higher complexity than string parsing, which is known to be $\mathcal{O}(n^3)$ (actually, even slightly better). Still there are parsers such as the DIAGEN parser [11] that perform very good. Functional-logic parsers unfortunately are not that efficient. Whereas the DIAGEN parser uses dynamic programming techniques, the functional-logic approach relies on backtracking. Nevertheless, it can be applied to non-toy examples also. For instance, a process hypergraph with 200 successive *act* hyperedges (created by the generator function `generateBPM`) can still be parsed in less than one second on standard hardware (2GHz, 2GB RAM). The used call has been:

```
BPM> findall (\s->process (generateBPM 200)=:s)
```

All examples throughout this paper have been executed with the Münster Curry Compiler², which generates comparatively efficient native code.

6 Conclusion

Using graph parser combinators, a solution for the (bidirectional) transformation of BPMs to BPEL has been realized in a straightforward way. BPMs are modeled as hypergraphs and the language is described by a context-free hyperedge replacement grammar, which directly can be translated into a parser. This case study has also shown the flexibility of graph parser combinators in the construction of a semantic representation. Not only can BPM graphs be derived backwards from a given BPEL structure, but it is also possible to complete partial BPMs or to use the parser for language generation.

² <http://danae.uni-muenster.de/~lux/curry/>

Unfortunately, only structured BPMs can be described by a context-free hypergraph grammar. So, the presented approach cannot be applied to arbitrary or even quasi-structured models in a straightforward way.

An installation of the framework with some examples is provided as a SHARE virtual machine at:

http://is.tm.tue.nl/staff/pvgorp/share/?page=ConfigureNewSession&vdi=Ubuntu-8.10_GB9_grappa-bpm.vdi

References

1. Object Management Group: Business Process Modeling Notation (BPMN) (2009) <http://www.omg.org/docs/formal/09-01-03.pdf>.
2. OASIS: Web Services Business Process Execution Language Version 2.0 (2007) <http://docs.oasis-open.org/wsbpel/2.0/wsbpel-v2.0.pdf>.
3. Mazanek, S., Minas, M.: Functional-logic graph parser combinators. In Voronkov, A., ed.: Proc. of the 19th International Conference on Rewriting Techniques and Applications. Volume 5117 of LNCS., Springer (2008) 261–275
4. Minas, M.: Hypergraphs as a uniform diagram representation model. In: TAGT'98: Selected papers from the 6th International Workshop on Theory and Application of Graph Transformations. Volume 1764 of LNCS., Springer (2000) 281–295
5. Hanus, M.: Curry: An Integrated Functional Logic Language (Version 0.8.2). (2006) Available at <http://www.curry-language.org/>.
6. Drewes, F., Habel, A., Kreowski, H.J.: Hyperedge replacement graph grammars. In Rozenberg, G., ed.: Handbook of Graph Grammars and Computing by Graph Transformation. Vol. I: Foundations. World Scientific (1997) 95–162
7. Hanus, M.: Multi-paradigm declarative languages. In Dahl, V., Niemelä, I., eds.: Logic Programming, 23rd International Conference, ICLP 2007, Porto, Portugal, September 8-13, 2007, Proceedings. Volume 4670 of LNCS., Springer-Verlag (2007) 45–75
8. Antoy, S., Echahed, R., Hanus, M.: A needed narrowing strategy. *Journal of the ACM* **47**(4) (2000) 776–822
9. Mazanek, S., Minas, M.: 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 (2008) 55–66 Technical Report number 0811.
10. Caballero, R., López-Fraguas, F.J.: A functional-logic perspective of parsing. In Middeldorp, A., Sato, T., eds.: Functional and Logic Programming, 4th Fuji International Symposium, FLOPS99, Tsukuba, Japan, November 11-13, 1999, Proceedings. Volume 1722 of LNCS., Springer-Verlag (1999) 85–99
11. Minas, M.: Concepts and realization of a diagram editor generator based on hypergraph transformation. *Science of Computer Programming* **44**(2) (2002) 157–180

A Module Grappa

Below, the *complete* Grappa library is given. This little code is sufficient already for the realization of powerful parsers. How it works is described in [3]. There also is an advanced version of the library with even better support for language generation and completion [9]. However, this is omitted here to keep things simple.

```
module Grappa where

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

type Grappa t res = Graph t -> (res, Graph t)

-- primitive parser, always succeeds without consuming any input
pSucceed :: res -> Grappa t res
pSucceed v g = (v, g)

-- primitive parser, consumes a single edge
edge :: Edge t -> Grappa t ()
edge e g | g:=:(g1++e:g2) = ((), g1++g2)
      where g1, g2 free

-- sequential composition of p and q
(<*>) :: Grappa t (r1 -> r2) -> Grappa t r1 -> Grappa t r2
(p <*> q) g = case p g of
  (pv, g') -> case q g' of
    (qv, g'') -> (pv qv, g'')

-- application of a function to the result of a parser
(<$>) :: (res1->res2) -> Grappa t res1 -> Grappa t res2
f <$> p = pSucceed f <*> p
-- replace result of a parser by a constant value
(<$) :: res1 -> Grappa t res2 -> Grappa t res1
f <$ p = const f <$> p

-- sequential composition where the result of q is disregarded
(<*) :: Grappa t res1 -> Grappa t res2 -> Grappa t res1
p <* q = (\x _ -> x) <$> p <*> q
-- sequential composition where the result of p is disregarded
(*>) :: Grappa t res1 -> Grappa t res2 -> Grappa t res2
p *> q = (\_ x -> x) <$> p <*> q
```

B Generating BPEL XML from BPEL terms

The following Curry code transforms a BPEL term to a BPEL-XML string.

```
bpel2xml :: BPEL -> String
bpel2xml f = "<process>\n" ++
            seq2xml f ++
            "</process>\n"

seq2xml [e] = bpelComp2xml e
seq2xml es@(e1:e2:_) = "<sequence>\n" ++
                      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" ++
                            seq2xml f1 ++
                            seq2xml f2 ++
                            "</flow>\n"
bpelComp2xml (Switch c1 c2 f1 f2) =
    "<switch>\n" ++
    "<case cond=\"" ++ c1 ++ "\">\n" ++
    "    seq2xml f1 ++
    "</case>\n" ++
    "<case cond=\"" ++ c2 ++ "\">\n" ++
    "    seq2xml f2 ++
    "</case>\n" ++
    "</switch>\n"
```

For the example of Fig. 2 this eventually results in the following XML:

```
<process>
<flow>
<sequence>
<invoke name="act1"/>
<invoke name="act2"/>
<receive name="ev1"/>
</sequence>
<sequence>
<invoke name="act3"/>
<switch>
<case cond="cond1">
<sequence>
<invoke name="act4"/>
<wait name="ev2"/>
</sequence>
</case>
<case cond="cond2">
<invoke name="act5"/>
</case>
</switch>
</sequence>
</flow>
</process>
```