

Transforming BPMN to BPEL Using Parsing and Attribute Evaluation with respect to a Hypergraph Grammar

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. Given a BPM, a hypergraph parser can be used to construct its derivation tree. Finally, the corresponding BPEL code can be generated using attribute evaluation.

A diagram editor for structured BPMs has been realized using the `DIAGEN` framework. On request, this editor outputs the BPEL code corresponding to the user's diagram.

1 Introduction

This paper describes how BPMs [1] can be translated to BPEL code [2] using hypergraph parsing and attribute evaluation. 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 and unambiguous hypergraph grammar. This means, every syntactically correct diagram has a unique derivation tree. By extending this grammar with attribute evaluation rules, a string representation such as BPEL code can easily be generated.

For the generation of BPEL code from concrete diagrams an editor has been created using the `DIAGEN` [3] editor generator framework. `DIAGEN` uses hypergraphs as a diagram model and hypergraph parsing for syntax analysis, i.e. an efficient hypergraph parser is readily available. So, in this paper, the patterns, a structured BPM is constructed of, are represented as productions of a hypergraph grammar. The parser performs the decomposition without any additional specification effort besides the grammar.

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.

The corresponding BPM hypergraph is shown in Fig. 2. Hyperedges are represented as boxes and nodes as black dots. If a node is attached to an edge, it

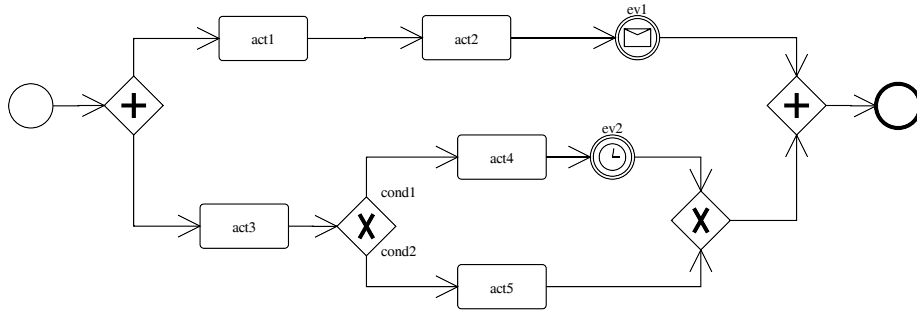


Fig. 1. Example process

is connected to this edge by a line, which is also called tentacle in this context. Tentacles are labeled to indicate the roles of the connected nodes. For instance, activities (edge label *act*) always have an incoming (*i*) and an outgoing tentacle (*o*); 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 (*l*), top (*t*), bottom (*b*), and right (*r*)).

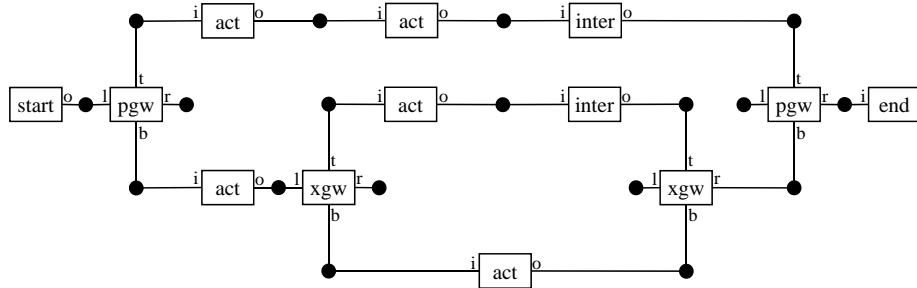


Fig. 2. BPM Hypergraph of Fig. 1

The hypergraph language of well-structured BPMs can be defined using a hyperedge replacement grammar (HRG). In *DIAGEN*, such grammars (more precisely an extended notion thereof) are used to define the abstract syntax of visual languages. They generalize the idea of Chomsky grammars for strings, which are used by standard compiler generators. The availability of derivation trees allows for a straightforward representation of the syntactic structure of a diagram.

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 clarity, loops are not considered. Ignore the program code right from the productions and the subscript numbers following some edge labels for a moment. Productions $L ::= R_1, L ::= R_2, \dots$ with the same left-hand side can be 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 *Elem*. The starting hypergraph consists of just a single *Process* edge.

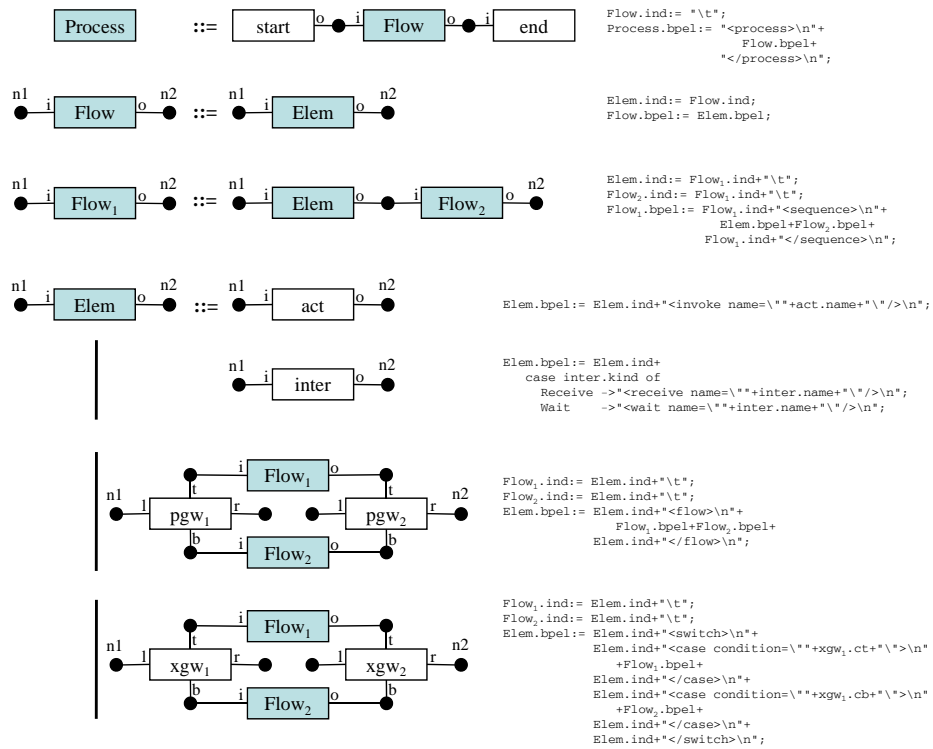


Fig. 3. (Attributed) productions of $G_{Process}$

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. [4]. 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.

The hypergraph parser used in DIAGEN constructs the derivation tree of a given hypergraph (if existing). An example hypergraph together with its derivation tree is shown in Fig. 4. How the parser actually works is described in [5].

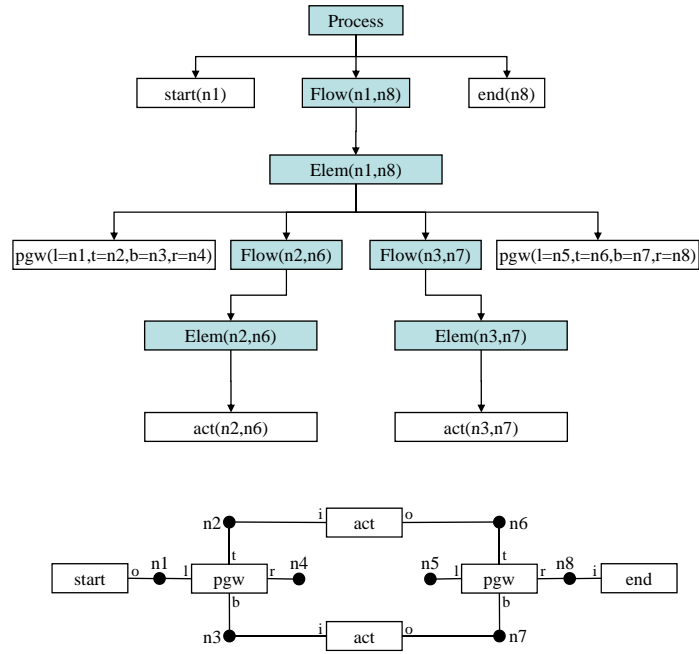


Fig. 4. Small BPM hypergraph and derivation tree thereof

DIAGEN directly supports the translation of a diagram into some domain-specific data structure (as required for the transformation of BPMs to BPEL). It provides attribute evaluation to this end, which is a well-known technique in the domain of compiler construction [6]: Each hyperedge carries some attributes. Number and types of these attributes, which have to be specified by the editor developer, depend on the hyperedge label. Productions of the hypergraph grammar may be augmented by attribute evaluation rules, which assign values to attributes of the edges involved. Both, inherited and synthesized attributes are supported.

The attribute evaluation rules for the transformation of BPM to BPEL can also be found in Fig. 3 (right-hand side). In evaluation rules, edge labels are used to refer to particular edges of the corresponding production. Subscript numbers have been added where a production contains more than one edge with the same label (similar to the notation used in [6]). The string attribute `bpe1` of the nonterminal hyperedges is a synthesized attribute carrying the BPEL code of the respective sub-diagram. To show the flexibility of the approach, also an inherited attribute (`ind`) has been added, which is used to determine the indentation level. Of course, also a standard XML pretty printer could have been used for this purpose. As usual, the special characters `\n` and `\t` used in Fig. 3 denote a line break and a tab stop, respectively, and the `+` operator is used for string concatenation.

```

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

```

Fig. 5. Generated BPEL code for BPM of Fig. 1

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 rules how attribute values are computed as soon as the values of others are known. The attribute evaluation mechanism of the editor 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; xgw hyperedges carry the conditions **ct** (top flow) and **cb** (bottom flow) of the corresponding exclusive gateway.

Fig. 5 shows the output resulting from the example process of Fig. 1. The whole XML file is determined by the value of the **bpel** attribute of the derivation tree's root edge.

3 Discussion and Related Work

Although hypergraph parsing generally has a higher complexity than string parsing, which is known to be $\mathcal{O}(n^3)$ (actually, even slightly better), the DIAGEN

parser performs very good on BPMs. A process hypergraph with more than 400 hyperedges can be parsed in less than 50 milliseconds on standard hardware (2GHz, 2GB RAM). Actually, in the developed DIAGEN editor not the parser, but rather the layouter is the performance bottleneck. However, layout can be switched off by the editor user.

Another important question is the scope of the proposed approach, i.e. which kinds of business process models can be treated. DIAGEN does not require a language to be context-free. So-called embedding productions are supported in addition to the context-free productions. Those allow to embed edges into some context. That way, all practical languages can be described (at least there is no counter-example yet). However, in absence of a comprising derivation tree, it is much more difficult to construct BPEL by attribute evaluation. So, BPEL generation for non-structured BPMs cannot be realized easily with DIAGEN.

At the moment the developed editor requires gateways to have exactly two branches. Support for three branches can be added by an additional grammar production. However, support for an arbitrary number of branches requires the whole language to be modeled in a different way. The reason is that gateway edges have only four tentacles whose roles are clearly determined. Fig. 6 shows how the language could be modeled to relax this restriction. A gateway now also has only an incoming and an outgoing tentacle. Each branch starts with a special *branch* hyperedge. This language can still be described by a context-free grammar. However, in order to keep this grammar unambiguous, successive branches have to be connected with each other to impose an order. Therefore, the tentacles upper (u) and lower (l) of the *branch* edges are required.

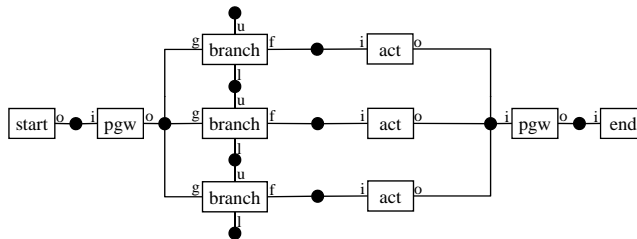


Fig. 6. Modeling BPMs with support for several branches

Related Work: For comparison, in [7] a pattern-based translation of BPMs to BPEL code has been proposed. In this paper, the patterns, a structured BPM is constructed of, are represented as productions of a hypergraph grammar. The parser performs the decomposition without any additional specification effort besides the grammar. Most other approaches for the transformation of BPMs to BPEL rely on the identification of so-called single-entry single-exit regions (SESE) in a BPM. In the hypergraph grammar $G_{Process}$ the nonterminal *Flow* captures exactly this notion. Derivation trees and parsing also have been used

already in other BPM tools. For instance, the special-purpose parser proposed in [8] even runs in linear time. However, the present approach is completely generic, i.e. a meta-tool has been used. No additional programming effort has been required at all.

4 Conclusion

Using the DIAGEN framework, a solution for the transformation of BPMs to BPEL has been realized in a straightforward way. It is based on parsing and attribute evaluation with respect to hypergraph grammars. The proposed solution not only comprises the transformation part though. As a byproduct, a fully-fledged diagram editor for business process models has been developed. This editor has not been described in this paper though.

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. Those, however, are already known to be more error-prone than structured ones [9].

A screencast of the developed editor and an executable jar of the editor are provided at <http://www.unibw.de/inf2/DiaGen/bpmn2bpe1/>. A SHARE virtual machine is also provided: http://is.tm.tue.nl/staff/pvgorp/share/?page=ConfigureNewSession&vdi=Ubuntu-8.10_GB9_diagen-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. Minas, M.: Concepts and realization of a diagram editor generator based on hypergraph transformation. *Science of Computer Programming* **44**(2) (2002) 157–180
4. 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
5. Bardohl, R., Minas, M., Taentzer, G., Schürr, A.: 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 Publishing Co., Inc., River Edge, NJ, USA (1999) 105–180
6. Knuth, D.E.: Semantics of context-free languages. *Theory of Computing Systems* **2**(2) (1968) 127–145
7. Ouyang, C., Dumas, M., ter Hofstede, A., van der Aalst, W.: Pattern-based translation of BPMN process models to BPEL web services. *International Journal of Web Services Research* **5**(1) (2007) 42–62
8. Vanhatalo, J., Völzer, H., Koehler, J.: The refined process structure tree. In: *BPM '08: Proc. of the 6th Int. Conf. on Business Process Management*. Volume 5240 of LNCS., Springer-Verlag (2008) 100–115
9. Gruhn, V., Laue, R.: What business process modelers can learn from programmers. *Science of Computer Programming* **65**(1) (2007) 4–13