Chart generation using production systems

Sebastian Varges
Information Technology Research Institute
University of Brighton
Sebastian.Varges@itri.brighton.ac.uk

Abstract

Productions systems, traditionally mainly used for developing expert systems, can also be employed for implementing chart generators. Focusing on bottom-up chart generation, we describe how the notions of chart algorithms relate to the knowledge base and Rete network of production systems. We draw on experience gained in two research projects on natural language generation (NLG), one involving surface realization, the other involving both a content determination task (referring expression generation) and surface realization. The projects centered around the idea of 'overgeneration', i.e. of generating large numbers of output candidates which served as input to a ranking component. The purpose of this paper is to extend the range of implementation options available to the NLG practitioner by detailing the specific advantages and disadvantages of using production systems for NLG.

1 Introduction

A general question faced by the NLG practitioner is whether to use an off-the-shelf generator or develop one 'from scratch.' Often, understanding the workings of the off-the-shelf generator and providing the required input structures requires substantial work. Furthermore, the off-the-shelf generator may not provide the required functionality without additional programming, the project may dictate the use of a particular programming language or frequent interaction with components written in that language.

In this paper, we describe how to implement chart generators in production systems, i.e. from scratch. Production systems have traditionally mainly been used for developing expert systems [Giarratano and Riley, 1993]. The particular production system we use, JESS [Friedman-Hill, 2003] is implemented in Java, making it easy to interact with other Java components. The rule encodings described in this paper should also work for other production systems such as CLIPS ("C Language Integrated Production System", [Riley, 1999]).

We used JESS in two projects centering around the concept of overgeneration-and-ranking. Both involved the generation of a large number of alternative outputs which served as input to a ranker written in Java. Search was directed by influencing the agenda ordering of the chart generator. At the syntactic level, realization in both projects was 'shallow.' However, since we were using expert system technology, we were able to use more sophisticated domain reasoning for content planning – more specifically, referring expression generation – to drive the realizer in one of the projects.

The two characteristics just described, the ability to deal with issues of search and the integration with reasoning capabilities, make generation using production systems quite different from other methods of shallow generation. For example, pipelines of XSLT stylesheets can be used to transform XML trees encoding linguistic structures [Wilcock, 2001; Moore *et al.*, 2004]. However, the focus in using XSLT for generation is more on pursuing a single alternative than on searching for the best one. Furthermore, rewriting XML trees with XSLT lends itself toward top-down generation, whereas the use of production systems naturally results in bottom-up generation.

In the following section, we relate chart algorithms to production systems at a more abstract level. We then discuss two generators implemented in the production system JESS.

2 Chart algorithms and production systems

Knowledge-based expert systems consist of rules (or 'productions') and a knowledge base of facts (KB). Rules express logical implications and facts contingent truths. Facts match (possibly partially defined) fact descriptions, or 'conditions', in the antecedent of rules, and if all conditions required in a rule antecedent are satisfied, the consequent follows. In a production system, this means the production is 'activated.' All activated rules are collected in a 'conflict set.' After this, a decision is made about what rule to fire according to the search strategy (breadth-first and depth-first are usually built-in strategies). Processing can be thought of as a sequence of cycles where each cycle is defined as adding one fact to the KB, matching it against the rules, and computing what rules to fire next. This may result in new facts, and a new cycle begins.

The basic outline of knowledge-based expert systems is similar in spirit to declarative rule systems based on the logic programming paradigm in NLP in which the order of rule execution does not affect the result. The knowledge base serves as a store for proven facts and is hence comparable to the chart data structures in NLP. More precisely, the knowledge base corresponds to a passive chart since it only contains completed chart edges. Executing a rule consequent can involve the assertion or retraction of facts to/from the KB. Since a chart is monotonically accumulating, grammar rules only assert new edges in our implementations.

Another way of thinking about production systems is in terms of a blackboard architecture: the KB serves as a blackboard and rules/productions are triggered by new facts that are added to it.

Probably the most distinctive property of production systems is the compilation of the productions into a network. The Rete networks of production systems ('rete' is Latin for 'net') exploit structural similarities between rule antecedents by creating a network that allows facts to match antecedents in several rules at once. The idea is to create tests for conditions in rule antecedents and to share the results between rules [Forgy, 1982]. Matching a fact against the Rete network can partially 'activate' a whole set of rules that share that node. This avoids cycling over possibly large numbers of rules in turn and repeatedly matching the same elements.

Standardly, a forward chaining interpreter is used to execute productions. This corresponds to bottom-up tree traversal algorithms in computational linguistics terminology. The 'conflict resolution strategy' addresses the above mentioned question which of the activated rules should fire. It corresponds to the agenda strategy in chart parsing/generation. The following table relates the terminologies:

Production Systems	NLP
productions in Rete network	grammar rules
working memory/ knowledge base	passive chart
facts in knowledge base	passive edges
partially activated productions (inside Rete network)	active edges
conflict resolution strategy	agenda with ordering function

Since the partially activated productions are part of the Rete network, they are more difficult to observe in practice than the active edges of a chart. However, production systems such as JESS provide means to inspect the network.

3 Case study 1: realization with automatically derived rules

The first project involves overgeneration (and ranking) with rules that were extracted from a semantically annotated subset of the Penn Treebank II. The input to the realizer consists of a bag of semantic tags with associated values in the management succession domain, for example PERSON=Piere Vinken, AGE=61 or POST=chairman.[1]

We use structured (or 'unordered') facts of attribute-value pairs to define chart edges. Figure 1 shows a slightly simplified production that, given two facts with heads NP-POST_DESCR_ADJ and PP-POST_NODET generates a new fact with head VP-POST_DESCR_ADJ:

```
(defrule phrasal-rule-83
  (NP-POST_DESCR_ADJ (idx ?i0) (coidx ?cx0) (syn ?s0)
                     (consumes $?c0) (terms ?t0)
                     (instances ?table0) (deriv $?d0))
  (PP-POST_NODET (idx ?i1) (coidx ?cx1) (syn ?s1)
                 (consumes $?c1&:(set(create$ $?c0 $?c1)))
                 (terms ?t1) (instances ?table1)
                 (deriv $?d1))
  (phrasal-rule-83)
 =>
  (assert
    (VP-POST_DESCR_ADJ
        (idx (bind ?idx (npt))) (coidx ?cxc)
        (syn VP) (consumes (create$ $?c0 $?c1))
        (terms  was named to ?t0 ?t1)
        (instances (combine-tables ?table0 ?table1))
        (deriv [VP p83-0366 ?idx was named to $?d0 $?d1])
        (fired-by phrasal-rule-83))))
```

Figure 1: Grammar rule encoded as production

This production effectively results in the bottom-up traversal of a local tree. The names of the fact heads combine syntactic and semantic information. The facts have slots such as idx, a unique edge identifier, and deriv, which represents the derivation tree. Slot values are mainly only picked up by variables (prefixed by ? and $?) and passed on to the rule consequent. However, there is an exception: the condition matching the second edge with head PP-POST_NODET performs a test in order to prevent combinations of edges that express overlapping semantics, based on semantic indices associated with the attribute-value pairs of the input. These are stored in the consumes slot.

Typically, the facts matched on the left-hand side of the production shown above provide realizations such as 'the additional post' and 'of chief executive officer.' The newly generated edge combines this into 'was named to the additional post of chief executive officer.' Phonological forms are handled in the terms slot where they are represented in combination with semantic tags, for example 'the [POST_DESCR_ADJ additional] post.' These

[1] These semantic labels are slightly simplified. For reasons of space we cannot discuss the semantic annotation and rule construction here. See [Varges, 2003] for more details.

rules/productions encode a phrasal generator with shallow syntax: they only use syntactic categories. However, other simple syntactic features, for example for modeling number/gender/person agreement, could be incorporated by adding the appropriate slots and values.

The `instances` slot provides an example of the tight integration of the production system JESS with Java. The slot contains references to Java objects that are relevant to the ranker. Furthermore, `combine-instance-tables` on the right-hand side of the production is a function call that is simply passed on to the ranker written in Java.

The left-hand side of the example production also contains a condition that matches a simple unique fact for the name of the rule (`phrasal-rule-83`). In this way, rules can be dynamically blocked and unblocked. Only those rules are able to fire whose name has been asserted into the KB. Matching the name fact last in the rule antecedent allows the Rete compiler to identify possible common prefixes of match patterns across rules. This would not be possible if a unique match condition was placed first.

The realizer uses 476 automatically constructed productions. Sharing in the Rete network is mostly limited to the ‘pattern network.’ In contrast, the ‘join network’ is not able to reduce the number of match computations substantially because most grammar rules are binary (see [Varges, 2003] for more details). The realizer is capable of generating several hundred sentences within a few seconds. Averaged over 40 inputs taken from a test set, it produced 350 sentences within 5 seconds.

4 Case study 2: referring expression generation

The second use of a production system for NLG is for a manually written referring expression generator developed in the TUNA[2] project [Varges, 2004]. Conceptually, it consists of two modules: a reasoner that produces logical forms (descriptions of domain objects) from a domain representation and a realizer for those logical forms. Both modules are interleaved in the sense that the reasoner ‘marks’ logical forms that it is able to realize, and the domain reasoner is only allowed to combine simpler logical forms into more complex ones if they have been realized successfully. One way to describe this processing architecture is in terms of two interleaved chart algorithms that exchange chart items. The other is in terms of a blackboard architecture – in fact, the KB almost acts as whiteboard [Cahill *et al.*, 1999]. This is particularly intuitive here, and also closer to the implementation: the reasoner automatically responds to the marking of a logical form fact as being realized since productions are activated whenever a matching fact is added to the KB (or changed). In other words, the productions of the modules communicate via the KB.

The modules of the referring expression generator, i.e. reasoner and realizer, are implemented by means of namespaces for the facts and productions they contain. In addition, we define namespaces for facts representing the domain model and for the lexicon. We show some facts of the domain model since this is the starting point of the computation:

```
(DOMAIN::vertex (index v1))
(DOMAIN::type (name musician) (index v1))
(DOMAIN::attribute (name hair-colour) (value black)
                                      (index v1))
(DOMAIN::vertex (index v2))
(DOMAIN::type (name instrument) (index v2))

(DOMAIN::relation (name hold) (rel-index r1))
(DOMAIN::out-relation (out v1) (rel-index r1))
(DOMAIN::in-relation (in v2) (rel-index r1))
```

Facts define object types, attribute-values pairs and relations between the objects. In the example above, they describe a musician with black hair holding an instrument. The facts are defined in namespace `DOMAIN` and are related by means of the indices held in the `index` and other slots. The use of `vertex` facts indicates that the representation is inspired by the Graph approach to referring expression generation [Krahmer *et al.*, 2003].

At the first stage of processing, content determination rules produce logical forms paired with a list containing the vertices of the domain objects they describe:

```
(LF::type-extension (extension v1) (id 3)
                    (lf "(" type "=" musician ")")
                    (depth 1) (type musician))

(LF::neq-type-extension (extension v1) (id 17)
        (lf "(" NOT   "(" type "=" instrument ")" ")")
        (depth 2) (negated 3))
```

The first fact lists, in slot `extension`, all domain objects of type ‘musician’, which is only `v1` in our example. The second fact contains the vertices of all objects that are not of type instrument, which again is `v1`. The facts contain the logical form as a sequence of atoms and strings. However, the fact heads are more important for matching since the logical forms (and many other slot values) are just passed on to the right-hand side. Facts can contain slots that are only relevant to their particular type, for example `negated` for facts with head `neq-type-extension`.

The namespace of the realization module is populated with facts that contain syntactic information and surface forms:

```
(REALIZER::np (phon not an instrument)
              (id 35) (dtr-left 5) (dtr-right 11)
              (num sing) (pers 3) (form neq-indefinite))

(REALIZER::syntax-semantics-mapping
              (sem-id 17) (syn-id 35))
```

The `REALIZER::np` fact above is the realization of a corresponding description fact in the `LF` namespace. The separate `REALIZER::syntax-semantics-mapping` fact records which description fact in the `LF` namespace is realized by which fact in the `REALIZER` namespace, again by means of indices. An alternative to using a mapping fact is to modify the `REALIZER::np` fact directly.

[2] EPSRC grant GR/S13330/01

The implementation currently consists of 95 productions and 71 additional functions that are invoked on the right-hand sides of the productions. These functions perform tasks such as counting domain objects and relations between objects, and computing the agenda ordering of the chart generator.

5 Discussion and conclusions

We consider the use of a production system in these two projects successful. Production systems are suitable for the efficient exploration of alternatives and for robust, 'data-driven' bottom-up processing. Such processing is particularly robust if 'flat' input structures are used, and this in turn is encouraged by the unnested structure of the facts in the knowledge base. This points to a characteristic of production languages that at the same time is a source of their efficiency (by allowing the construction of the Rete networks) and a limitation: since the slot values of facts cannot contain recursive data structures, we need to resort to the use of indices to express that certain facts 'belong together.' This is evident in the index slots of the domain model in the second case study, for example. The same technique is used in the NL-SOAR project [Rubinoff and Lehman, 1994], to our knowledge the most extensive use of production systems for NLP. However, if indices are used extensively, more work needs to be done in the join network part of the Rete network, partially offsetting its benefits. In sum, we see the following advantages of using production systems for NLG:

- they are able to deal with large numbers of (possibly machine-learned) rules (case study 1, see also [Doorenbos, 1993]),
- they are suitable for integrating NLG with more general inferencing/reasoning (case study 2),
- general advantages: (largely) declarative behaviour; seamless integration with Java if JESS is used; development: rapid prototyping, read-eval-print loop.

On the other hand, we see the following disadvantages:

- facts of limited structure and unavailability of unification: production systems are not well-suited for developing generators based on unification-based grammar formalisms such as HPSG;
- general disadvantages: alternative tree-traversal strategies such as head-driven approaches not straightforward to implement; development: limited compile-time checks.

One possibility of addressing the lack of nested data structures might be to compile complex feature formalisms into a more shallow cfg-like format, i.e. again to automatically generate productions (as we did in the first project). A further avenue of future work is the pre-compilation of descriptions for the referring expression generator. This is motivated by the fact that in our approach a large part of the computation of descriptions is independent of the specific generation input. Such precompilation should result in significant efficiency gains.

In this paper we have not been able to explore production systems and, in particular, their Rete networks in full detail. However, we hope to have convinced the reader that the use of production systems for NLG can be advantageous when relatively shallow but rule-based generation capabilities are required.

References

[Cahill *et al.*, 1999] Lynne Cahill, Christy Doran, Roger Evans, Chris Mellish, Daniel Paiva, Mike Reape, Donia Scott, and Neil Tipper. In Search of a Reference Architecture for NLG Systems. In *Proc. of EWNLG-99*, 1999.

[Doorenbos, 1993] Robert B. Doorenbos. Matching 100,000 learned rules. In *Proc. of AAAI-93*, 1993.

[Forgy, 1982] Charles L. Forgy. Rete: A Fast Algorithm for the Many Pattern/ Many Object Pattern Match Problem. *Artificial Intelligence*, 19:17–37, 1982.

[Friedman-Hill, 2003] Ernest Friedman-Hill. *JESS - the Java Expert System Shell, Version 6.x.* Sandia National Laboratories, Software available at http://herzberg.ca.sandia.gov/jess/, 2003.

[Giarratano and Riley, 1993] Joseph Giarratano and Gary Riley. *Expert Systems: Principles and Practice.* PWS Publishing, Boston, 2nd edition, 1993.

[Krahmer *et al.*, 2003] Emiel Krahmer, Andre Verleg, and Sebastiaan van Erk. Graph-based Generation of Referring Expressions. *Computational Linguistics*, 29(1):53–72, 2003.

[Moore *et al.*, 2004] Johanna Moore, Kaska Porayska-Pomsta, Sebastian Varges, and Claus Zinn. Generating tutorial feedback with affect. In *Proc. of FLAIRS*, 2004.

[Riley, 1999] Gary Riley. CLIPS: A Tool for Building Expert Systems. http://www.ghg.net/clips/CLIPS.html, 1999.

[Rubinoff and Lehman, 1994] R. Rubinoff and J. F. Lehman. Real-time Natural Language Generation in NL-Soar. In *Proc. of IWNLG*, 1994.

[Varges, 2003] Sebastian Varges. *Instance-based Natural Language Generation.* PhD thesis, ICCS, School of Informatics, University of Edinburgh, 2003.

[Varges, 2004] Sebastian Varges. Overgenerating referring expressions involving relations. In *Proc. of INLG-04*, 2004.

[Wilcock, 2001] Graham Wilcock. Pipelines, Templates and Transformations: XML for Natural Language Generation. In *Proc. of First NLP and XML Workshop (NLPRS-2001)*, 2001.