Deutsches
Forschungszentrum
für Künstliche
Intelligenz GmbH

Research
Report
RR-92-43

A Heuristic Driven Chart-Parser for Attributed Node Labeled Graph Grammars and its Application to Feature Recognition in CIM

Christoph Klauck, Jakob Mauss

September 1992

Deutsches Forschungszentrum für Künstliche Intelligenz
GmbH

Postfach 20 80
67608 Kaiserslautern, FRG
Tel.: + 49 (631) 205-3211
Fax: + 49 (631) 205-3210

Stuhlsatzenhausweg 3
66123 Saarbrücken, FRG
Tel.: + 49 (681) 302-5252
Fax: + 49 (681) 302-5341

A Heuristic Driven Chart-Parser for Attributed Node Labeled Graph Grammars and its Application to Feature Recognition in CIM

Christoph Klauck, Jakob Mauss

DFKI-RR-92-43

A short version of this paper will be published in the Proceedings of the International Workshop on Structural and Syntactic Pattern Recognition (SSPR'92).
This paper was also accepted in a short version ba the V International Symposium on Artificial Intelligence (V ISAI'92).

This work has been supported ba a grant from The Federal Ministry for Research and Technology (ITWM-8902 C4).

A Heuristic Driven Chart-Parser for Attributed Node Labeled Graph Grammars and its Application to Feature Recognition in CIM

Jakob Mauss and Christoph Klauck

DFKI GmbH, Projekt ARC-TEC,
P. O. Box 2080, D-6750 Kaiserslautern
Phone: ++49-631-205-3477 Fax: ++49-631-205-3210,
e-mail: klauck@dfki.uni-kl.de

Abstract

To integrate CA*-systems with other applications in the CIM world, one principal approach currently under development is the feature recognition process based on graph grammars. It enables any CIM component to recognize the higher-level entities - the so-called *features* - used in this component out of a lower-data exchange format, which might be the internal representation of a CAD system as well as some standard data exchange format. In this paper we present a 'made-to-measure' parsing algorithm for feature recognition. The heuristic driven chart based bottom up parser analyzes attributed node labeled graphs (representing workpieces) with a (feature-)specific attributed node labeled graph grammar (representing the feature definitions) yielding a high level (qualitative) description of the workpiece in terms of features.

Contents

List of Figures

1 Motivation

Research in feature-based CA*-systems like CAD (Computer Aided Design), CAPP (Computer Aided Process Planning) or CAM (Computer Aided Manufacturing), has been motivated by the realization that geometric models represent a workpiece in greater detail than can be utilized e.g. by a designer, strength calculator or process planner. When CA*-experts view a workpiece, they perceive it in terms of their own expertise. These terms, the so-called *features*, which are build upon a *syntax* (shape description: topological graph containing geometry and technology) and a *semantics* (description of related informations, e.g. skeletal plans in manufacturing or functional relations in design), provide an abstraction mechanism to facilitate e.g. the creation, manufacturing and analysis of workpieces. Features that are required e.g. for design may differ considerably from those required e.g. for manufacturing or assembly, even though they may be based on the same geometric and technological entities (cf. [FiSa 90]).

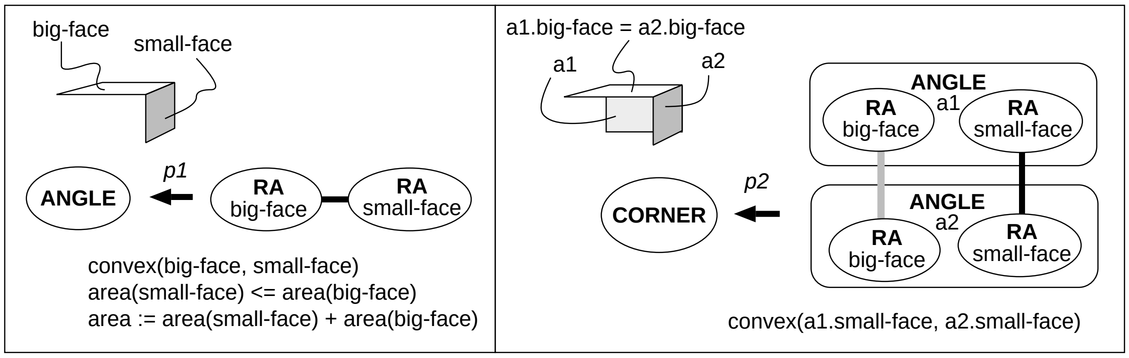
big-face
small-face
ANGLE
p1
RA big-face
RA small-face
convex(big-face, small-face)
area(small-face) <= area(big-face)
area := area(small-face) + area(big-face)
a1.big-face = a2.big-face
a1
a2
ANGLE
a1
RA big-face
RA small-face
CORNER
p2
ANGLE
a2
RA big-face
RA small-face
convex(a1.small-face, a2.small-face)

Figure 1: Two sample feature definitions

So representing features and recognizing them out of the (lower-level) workpiece description is a necessary step to bridge the gap between the several CA*-systems and an important step towards truly Computer Integrated Manufacturing (CIM). The expected advantages of a close coupling of CA*-systems are: The information interchange shall lead to better knowledge transfer, to shorter turnaround times and to improved feedback. In the end, higher flexibility and generally better results are expected.

In current research one method to represent and recognize features is based on graph grammars (cf. [FiSa 90, ChHe 91]). This area is a well established field of research and provides a powerful set of methods like parsing and knowledge about problems, their complexity and how they could be solved efficiently. The use of graph grammars for feature descriptions facilitates the application of these results to the area of feature recognition. So, in consideration of the feature characteristics, 'made-to-measure' tools must be developed to make the recognition and representation process very efficient.

From this point of view we present in this paper a heuristic driven chart based parser for parsing attributed node labeled graphs representing workpieces with attributed node labeled graph grammars representing the feature definitions. Parsing yields a high level (qualitative) description of the workpiece in terms of the so-called features to support e.g. a feature based CAPP-system (cf. [Cha 90a, Cha 90b]). The nodes of a workpiece graph represent geometric primitive surfaces, the node label decode the type of the surface, the attributes carry detailed geometric and technologic information and the edges decode the topology of the workpiece, i.e. two nodes are adjacent if the corresponding surfaces touch each other.

Figure 1 shows two simple features and its associated productions. An ANGLE consists of two connected atomic features (surfaces), called big-face and small-face within the ANGLE feature. Several constraints are specified, like *convex(big-face, small-face)* and a function is used to define the ANGLE attribute area.

The complex feature CORNER is defined by two overlapping ANGLE features. The two ANGLEs (a1 and a2) overlap with their big-face, say they share the same surface as big-face. Additional their small-faces must be neighboring. The grammar formalism introduced in the next section enables us to express two kinds of deep relations between the components of nodes (black: neighboring, grey: overlapping).

To become more acquaint with the effect of feature characteristics to our formalism, we briefly introduce now the term feature and the most important characteristics of its definition. Detailed description and the analogy to graph grammars can be found e.g. in [Klau 91].

We define the term **feature** as a description element based on geometrical and technological data of a product which an expert in a domain associates with certain informations.

Some of the syntactical characteristics of features are:

1. *Interaction:* Areas of features can overlap (see figure 1). This must be taken into account by the formalism and by the parser.
2. *Dependence of Dimensions:* In dependence of the dimensions, the same structures may be identified as different features. This is expressable by functional constraints on attributes included in a feature definition.
3. *Hierarchy:* A complete feature description of a workpiece forms a hierarchical structure of features: the output of our parser.
4. *Qualitative Description:* To describe a feature an expert uses only less geometrical and technological informations; he uses a qualitative description.
5. *Ambiguity:* A feature can often be derived in many syntactic different but semantic equivalent ways. We are interested in only one pregnant description of the workpiece.

6. *Similar Features:* Features may be similar with respect to a has-parts or a is-a hierarchy. A terse formulation and an efficient treatment of such variants and special cases is desired. In this work, the recognition of variants is supported by using a chart. The treatment of special cases is simplified by a close coupling of parsing and taxonomic reasonig.

The above characteristics lead us to a 'made-to-measure' grammar formalism, called ANLGG (Attributed Node Labeled Graph Grammar) introduced in the next section.

2 Attributed Node Labeled Graph Grammars

In this section we will briefly define the terminology of attributed node labeled graph grammars as used in this paper, shortly called ANLGG. Surveys and detailed introduction to graph grammars can be found e.g. in [Ehr 79-90].

A **node** in our formalism ANLGG is a structure with two kinds of slots, called **role**[1] and **attribute**. A node can be seen as a partial function mapping slots to their corresponding values. An attribute value is an arbitrary atomic object, a role value is always a node. Let **cast**($\boldsymbol{v}$) be the set of all role values of the node v, and cast$^*(v)$ be the transitive closure of the function cast, i.e. the set of all role values of v and role values of role values of v and so on, including v. The nodes in cast(v) are assigned to v by an application of a graph grammar production as explained later in this section. There may be edges between the nodes in cast(v); therefor the node v can also be seen as a graph. The nodes in cast$^*(v)$ are called **components** of v and represent a hierarchy of features as mentioned in the previous section with v serving as root-node. The leaves (atomic features) of the hierarchy are given by the nodes of a terminal graph (representing e.g. a workpiece).

A **path** in a role r_0 is a finite sequence $\pi = [r_0 \ldots r_n]$ of roles and $v(\pi)$ denotes the component v_n of v where $v_0 := v(r_0), v_1 := v_0(r_1) \ldots v_n := v_{n-1}(r_n)$. The last element in π may be an attribute, in this case $v(\pi)$ denotes the corresponding attribute value. Occasionally we use a superscript notation like in v^* to indicate, that v^* is a component of v. For a given set V of nodes let V^* denote the union $\bigcup_{v \in V} \mathrm{cast}^*(v)$.

A finite undirected node labeled graph graph, or **graph** for short is a 4-tuple $g := (V, E, S, \varphi)$ where

[1]The notion of *role* as used here is inspired by the notion of roles in KL-ONE like frameworks. Like in KL-ONE, role values will be restricted to instances of a given sort or to one of its subsorts. Additional, roles in our framework carry an implicit (1,1) number restriction (i.e. there's only one filler at a time).

V	is a finite set of nodes, $\|V\|$ is the number of nodes in g
E	$\subseteq V^* \times V^*$ is a set of undirected edges
S	is a finite nonempty alphabet of node labels
φ	is a labeling function $\varphi : V^* \to S$

For a given node v the label $\varphi(v)$ is called the **sort** of v. Note, that E and φ are defined for $V^* \supseteq V$, so the nodes itself possess graph structure.

For example the graph g_1 in figure 2 below got two nodes $V = \{\text{ANGLE}_4, \text{RA}_3\}$ and four edges. The node ANGLE_4 got two roles *big-face* and *small-face* and an attribute *area*, $\text{cast}(\text{ANGLE}_4) = \{\text{RA}_1, \text{RA}_2\}$ and $\text{ANGLE}_4(\text{big-face}) = \text{RA}_1$ with $\varphi(\text{RA}_1) = \text{RA}$ (for RectAngular surface). $\text{ANGLE}_4(\pi) = (3, 0, 0)$ where π = [big-face direction-vec2] is a path in *big-face*. V^* of g_1 contains four nodes and the node ANGLE_4 can also be seen as a graph with two nodes RA_1 and RA_2 and one edge.

A **terminal graph** is a graph $g_0 = (V_0, E_0, S, \varphi_0)$ where $\forall v \in V_0 : \text{cast}(v) = \emptyset$ and metasort $(\varphi(v)) = \mathit{terminal}$. For example g_0 in figure 2 is a terminal graph, but g_1, g_2, and g_3 are not.

The sorts in S are structured by a partial order $<_S$, the **sort hierarchy**, a subsumption hierarchy. If $s_1 <_S s_2$ we say s_1 is a **subsort** of s_2. There are no assumptions or restrictions concerning inheritance, i.e. the relation between the roles and attributes of a sort and those of its subsorts. The parser will just treat any instance of s_1 like an instance of s_2, but not vice versa. Sort hierarchies are used here in order to reduce the number of productions needed to describe the domain.

In our paper an attributed node labeled (feature) **graph grammar**, or ANLGG for short, is a 6-tuple $gg := (S, <_S, P, R, \text{sort}, \text{metasort})$ where

S	is the same alphabet as above
$<_S$	is a partial order of S, the sort hierarchy
P	is a finite set of productions
R	is the finite set of all role specifications of productions in P
sort	is a total function sort: $P \cup R \to S$ and
metasort	is a total function metasort: $S \to \{\mathit{terminal}, \mathit{nonterminal}, \mathit{goal}\}$

A **production** is a finite, nonempty set $p = \{r\text{-}spec_1, \ldots, r\text{-}spec_n\}$ of role specifications, defining the right hand side of p. Each role specification $\in p$ specifies a node such that the whole p specifies a connected graph with certain relational and functional constraints holding between the nodes. The specified graph may be replaced by a single node v, i.e. the production is applied to a graph g_n leading to a reduced graph g_{n+1} by

- identifying n nodes $v_i \in V_n^*$ with $\varphi(v_i) \leq_S \text{sort}(r\text{-}spec_i)$, that satisfy the relational and functional constraints given by $r\text{-}spec_i$, $1 \leq i \leq n$,

- ◦ building V_{n+1} by removing these nodes from V_n and adding a new node v with $\varphi(v) \leq_S \text{sort}(p)$, that has one role r_i for each *r-spec*$_i$ with the corresponding removed node v_i serving as role value, i.e. $v(r_i) = v_i, \quad 1 \leq i \leq n$ and $V_{n+1} = \{v\} \cup V_n \setminus \text{cast}(v)$. The added v represents an instance of the left hand side of p.

Additional we need an embedding rule that specifies, what happens to the edges of the removed nodes $\{v_1, \ldots, v_n\} = \text{cast}(v)$ and how the added node v is connected to the remaining nodes $V_{n+1} \setminus \{v\}$ of the new graph g_{n+1}. We define:

- ◦ $E_{n+1} = E_n \cup \{(v, v_j^*) \mid \exists (v^*, v_j^*) \in E_n : \quad v^* \in \text{cast}^*(v) \text{ and } v_j^* \in V_n^* \setminus \text{cast}^*(v)\}$

This embedding simply states, that the added node shares an edge with all nodes, that have at least one connection with one of the removed nodes. Geometrical interpreted it states, that two composed objects touch each other, if at least two of their components do so. As a consequence of our embedding rule, the reverse application of a production (i.e. from left to right) is not uniquely defined.

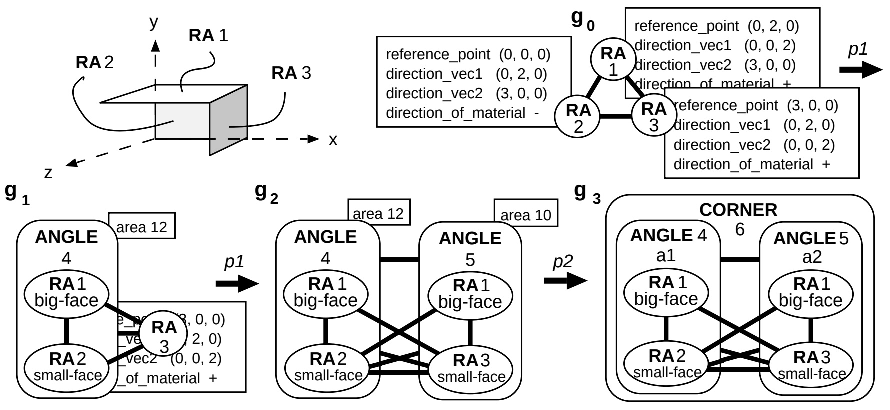

Figure 2: A simple workpiece graph and its derivation

A **role specification** is a 4-tuple r-spec $= (r, C^{\approx}, C^{=}, C^{func})$ where r is the specified role, $C^{\approx}$ and $C^{=}$ are finite sets of path constraints, and C^{func} is a finite set of functional constraints and predicates (see e.g. figure 1). A role may have several specifications within a grammar, but not within a production. A **path constraint** for a role r in a production p is a 2-tuple (π, π_0) of pathes and denotes two nodes v^* and v_0^* as follows: Let v be a node added by an application of p and $v_0 = v(r)$. Then $v^* := v(\pi)$ and $v_0^* := v_0(\pi_0)$. The path constraint enforces v^* and v_0^* to be connected or to be identical (i.e. to overlap) depending whether it occurs in $C^{\approx}$ or in $C^{=}$. The ability to express deep relations, i.e. relations between the components of the nodes in a graph enables us e.g. to express an alignment of components and thereby to reduce the ambiguity inherent to our grammar formalism.

A **functional constraint** of a r-spec of a production p is an expression of the form $a = f(\pi_1, \ldots, \pi_n)$, where a denotes an attribute value $v(a)$ of a node v added by p and f is applied to the (role or attribute) values $v(\pi_i)$. Analogous a **predicate** is an expression of the form $p(\pi_1, \ldots, \pi_n)$ and yields true when applied to the denoted values $v(\pi_i)$. A node $v_0 \in \text{cast}(v)$ satisfies a given r-spec of a role r if $\varphi(v_0) \leq_S$ sort(r-spec) and $v_0 = v(r)$ such that v satisfies all constraints and predicates given by r-spec.

For example, the two r-specs for big-face and small-face in production $p1$ in figure 1 contain two predicates and one functional constraint each (only depicted once in the figure). The r-spec for role a2 in production $p2$ contains a $C^=$ path constraint ([a1 big-face], [big-face]), and, corresponing, a1's r-spec got a path constraint ([a2 big-face], [big-face]) to specify a shared, overlapping component for a1 and a2.

Let g_0 be a terminal graph, $(g_0 \rightarrow g_1 \rightarrow \ldots \rightarrow g_n)$ a finite, nonempty sequence of successive applications of productions of a given gg such that g_n contains only one node v_g and metasort$(\varphi(v_g)) = goal$. Then that sequence is called a **derivation** of g_0 and the node v_g is called a **parse** or **feature tree** of g_0, where the nodes of this feature tree are given by cast$^*(v_g)$. Figure 2 shows an example of a derivation with a gg containing the two productions $p1$ and $p2$ given in figure 1.

3 The Graph Parsing Procedure

Before we describe the algorithm to derive all parses from a given terminal graph some special data structures and orders will be defined.

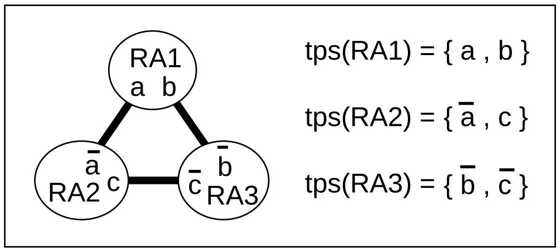
RA1
a b
$\bar{a}$
RA2 c
$\bar{c}$ $\bar{b}$
RA3
tps(RA1) = { a , b }
tps(RA2) = { $\bar{a}$, c }
tps(RA3) = { $\bar{b}$, $\bar{c}$ }

Figure 3: Representing edges by tiepoints

To handle edges more easy by the parser we associate with every edge $(v_1, v_2) = e \in E_0$ of a given terminal graph g_0 a 2-tuple of **tiepoints** $(tp, \overline{tp})$ corresponding to the two nodes v_1 and v_2 specifying the edge e. tps(v) defines the set of all tiepoints of the node v. If tp is a tiepoint then $\overline{tp}$ is its **complement** and vice versa. The embedding rule defined in the previous section introduces new edges at each application. But these new edges are defined in terms of the old ones, and depend by recursion on the set E_0 of edges of the initial terminal graph. The parser doesn't represent the added edges explicit, but calculates the connections from the tiepoints of the initial terminal graph using a global tiepoint-inheritance rule described below[2].

As a first heuristic extension to the given grammar we add to every production p a total order $<_p$. So p becomes an ordered sequence [r-spec$_1, \ldots,$ r-spec$_n$] of role specifications. The order decodes the strategy that the parser uses to build an instance of the production, i.e. he will try to find role values for the roles specified by p in the sequence given by $<_p$. The intention is to search for the most restricted role values first in order to effect an early pruning. If r-spec$_i <$ r-spec$_j$ and the parser can't find a role value for r-spec$_i$, he will never try to find a role value for r-spec$_j$. We constrain $<_p$ such that r-spec$_j$ specifies a connection or overlapping with at least one of the role values specified by r-spec$_i$, $1 \leq i < j$. This insures that the subgraph covered by a partial instantiation of a production is connected.

A **patch** is a partial (hypothesis, pp) or complete (fact, cp) instance of a production and consists of:

production	the production, whose partial or complete instance the patch is.
sort	the production's sort (or one of it's subsorts).
tps	the node's tiepoint set.
r-spec	the specification of the role that the patch is searching a role value for, empty, if the patch is complete.

Additional patches can have attributes and roles as defined above for nodes. The slot *sort* of a patch is treated like an attribute, i.e. it can be changed by a functional constraint to an appropriate subsort. We use a dotted notation, e.g. cp.tps denotes the tps of a complete patch cp. Two patches are **connected** if their tiepoint sets contain at least one complementary pair of tiepoints.

A cp is **combinable** with a pp iff

1) cp.sort $\leq_S$ sort(pp.r-spec),
2) For each path constraint $(\pi, \pi_0) \in$ pp.r-spec.C$^{\approx}$ the patches referenced by pp(π) and cp(π_0) are connected,
3) if $(\pi, \pi_0) \in$ pp.r-spec.C$^{=}$, then pp(π) = cp(π_0) and there are no further, unspecified overlaps between cp and pp,
4) and all constraints $\in$ pp.r-spec.C^{func} are satisfied in the sense defined above. The functional constraints are always satisfiable by making them true, i.e. by treating them as an assignment of the function value to the thereby defined attribute.

[2]It would have been possible, and perhaps more elegant, to define the embedding rule for ANLGGs wholly in terms of tiepoint-inheritance. We have chosen the somewhat awkward definition given in section 2 for reasons of compatibility with widespread graph grammar frameworks, as presented e.g. in [Ehr 79-90].

The **agenda** is a set of patches. The **chart** contains all patches not contained in the agenda. Its purpose is, to retrieve quickly the set of all patches, that are combinable with a given patch. Its task is not, to retrieve exactly this set but a likewise small, but complete superset of this. The superfluous patches can then be eliminated by evaluating the constraints. The retrieval of patches is done by applying the following four careful chosen access functions:

CP(tp,sort) $= \{\text{cp} \in \text{chart} \mid \text{cp.sort} = \text{sort} \quad \text{and} \quad \text{tp} \in \text{cp.tps}\}$
yields the set of all complete patches cp of sort *sort* contained in the chart that contain the tiepoint tp in its tps.

PP(tp,sort) $= \{\text{pp} \in \text{chart} \mid \text{sort(pp.r-spec)} = \text{sort} \quad \text{and} \quad \ldots\}$
yields a set of pp (not necessary all pp) of the chart, that are looking for a cp of sort *sort* as next role value such that the cp satisfies a connection constraint in pp.r-spec.$C^{\approx}$ if it contains tp in its tps. The function does not retrieve all pp with this property. However it is guaranteed that each cp combinable with a given pp contains at least one tp in its tps, such that for this tp pp $\in$ PP(tp,cp.sort). If the functions are implemented as (hash) tables this restriction avoids superfluous multiple entries of partial patches.

These functions may be implemented as an array of hash tables with tp serving as index and sort serving as key. The following two functions retrieve patches satisfying a given overlap constraint:

$\mathbf{CP}_{ov}(\mathbf{cp}^*)$ $= \{\text{cp} \in \text{chart} \mid \text{cp}^* \in \text{cast(cp)}\}$
yields all complete patches of the chart that contain cp^* as direct role value.

$\mathbf{PP}_{ov}(\mathbf{cp}^*)$ $= \{(\text{pp}, \pi_0) \mid \text{cp}^* = \text{pp}(\pi) \quad \text{and} \quad (\pi, \pi_0) \in \text{pp.r-spec.}C^{=}\}$
yields all 2-tuples (pp,π_0) of pp in chart such that pp is looking for a cp with an overlap $\text{cp}^* = \text{cp}(\pi_0)$ specified by pp.r-spec.$C^{=}$.

These functions may be implemented as hash tables with cp^* serving as key. Additional the chart contains two sets $\mathbf{CP}_0$ and $\mathbf{PP}_0$ containing all patches of the chart, whose tps is empty.

As a second and most important search guiding heuristic we add a total order $<_A$ for patches, its meaning will become clear below. The patch order $<_A$ may e.g. depend on the state of the agenda and an underlying ordering of the patch sorts or productions or attribute values or of a weighted combination of all. The weights may be determined by neural networks as proposed in [SuEr 90] or by genetic algorithms as proposed in [BGH 89].

Let us now describe the parsing procedure. The parser consists of three rules **initialize**, **choose** and **combine** that operate on two sets agenda and chart. The agenda is initialized with one patch for each of the graph's nodes and then choose and combine are applied alternately until the agenda runs empty. If this happens the chart i.e. CP_0 contains all possible parses (see figure 5). In practise, the parser will be stopped after the first parse is found.

```
parse graph:
     initialize chart and agenda
     until the agenda is empty do
          choose a best patch from agenda
          combine patch
          add patch to chart
     enduntil
```

`initialize` sets up an empty agenda and an empty chart, creates a cp for each node v of the input graph with cp.tps = tps(v) and adds it to the agenda.

`choose` picks up the most promising i.e. the $<_A$ greatest patch from the agenda, $<_A$ is completely free in ordering the patches, i.e. every order will finally lead to an empty agenda, however the workspace and time needed to encounter a first parse may crucially depend on the heuristic embodied by $<_A$.

```
combine patch:
     if patch is complete
        then propose patch
             continue-cp patch
        else continue-pp patch
     endif
```

`propose` combines a given cp with every production in `predict(cp.sort)` whose first r-spec is satisfied by the cp. For each such combination a new patch np is created with np.tps = cp.tps and is added to the agenda. For each sort $s \in S$ `predict(s)` $\subseteq P$ yields the set of productions $\{p \mid \ s \leq_S \text{sort}(\text{first}(p))\}$. These sets depend on the given graph grammar only and can be precalculated.

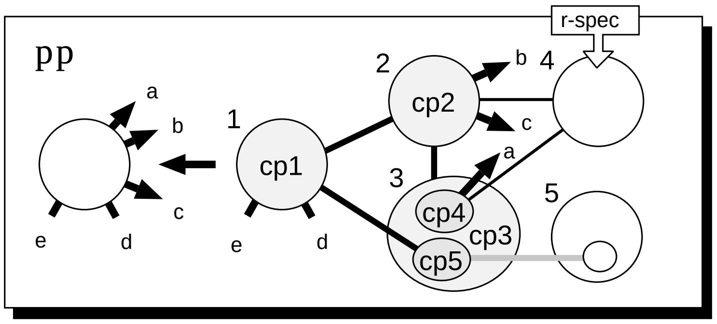

Figure 4: A partial patch with three filled roles

`continue-cp` combines a given cp with all possible pp and `continue-pp` combines a given pp with all possible cp contained in the chart creating a new patch for every successful cp-pp combination and adds it to the agenda. For a production with n roles n such binary combinations yield a cp that is a complete instance of the production. The intermediate pp are generally used several times to build alternative instances, so no derivation work has to be done twice[3].

The four chart access functions are used to retrieve candidates for the combination with a given patch, i.e. the candidates can be received by simple union and intersecting operations on the sets received by single (in the case of CP_{ov} recursive iterated) applications of these functions.

Take e.g. a look at figure 4: This pp is an instance of a production formed by 5 role specifications, values for the first three roles are already found. The pp is searching a role value for its fourth role r_4. The r-spec for this role states, that any candidate for the role needs a connection with r_2, occupied here by cp2 and a certain component of the value for the third role r_3, occupied here by cp4. The intersection of pp's tiepoint set pp.tps $= \{a, b, c, d, e\}$ with the tps of the two already determined neighbors of r_4 yields two sets of active tiepoints $\{a\}$ and $\{b, c\}$. Then the union $\cup_{s\in\mathrm{subsorts(sort(r_4))}}\mathrm{CP(a,s)} \cap (\mathrm{CP(b,s)} \cup \mathrm{CP(c,s)})$ yields the set of all cp in chart that satisfy all $\mathrm{C}^{\approx}$ constraints given for the fourth role.

In a similar fashion all other retrieval tasks can be done, using the four chart access functions defined above. E.g. for a given cp the set $\cup_{s\in\mathrm{supersorts(cp.sort)}} \cup_{\mathrm{tp}\in\mathrm{cp.tps}}$ $\mathrm{PP(tp,s)}$ contains all pp such that cp is of an appropriate sort and satisfys at least one of the $\mathrm{C}^{\approx}$ constraints given for pp's next role.

Note that a patch p1 and a candidate p2 retrieved from chart might overlap, i.e. before combining them it has to be tested that the intersection $\mathrm{cast}^*(\mathrm{p1}) \cap \mathrm{cast}^*(\mathrm{p2})$ contains exactly the set of overlappings specified by $\mathrm{C}^{=}$, even in the case of $\mathrm{C}^{=} = \emptyset$. The above intersection will not only contain eventually specified overlappings, but also all components of such overlappings. These components are of course legal, i.e. do not contribute to unspecified, forbidden overlappings.

The tps of a newly created patch np is the union of cp.tps and pp.tps with complementary pairs of tiepoints removed (i.e. without connections between cp and pp). Additional all tiepoints are removed that are contained in the tps of an overlapping component but not simultaneously in cp.tps and pp.tps. This is the tiepoint-inheritance rule mentioned above (and resembles the removal of complementary literals in resolution calculus).

For example, given the tiepoints from figure 3, the tps for the complete patch ANGLE_4 in figure 2 is calculated by:

$$\frac{\{b, a\}, \{\overline{a}, c\}}{\{b, c\}}$$

ANGLE_4 is connected to RA_3, since its tps $\{b, c\}$ shares at least one complemen-

[3]The combination of a pp with a cp corresponds roughly to the unification of two terms in unification-based formalisms (like D-PATR). But unlike unification our combination is nondestructive: The involved pp and cp are not modified and hence do not have to be copied before combination. This allows an efficient implementation.

tary pair of tiepoints with RA$_3$'s tps $\{\overline{b}, \overline{c}\}$. The tps of ANGLE$_5$ is calculated in the same manner:

$$\frac{\{a, b\}, \{\overline{b}, \overline{c}\}}{\{a, \overline{c}\}}$$

Finally, CORNER$_6$ inherits its tiepoints from ANGLE$_4$ and ANGLE$_5$:

$$\frac{\{b, c\}, \{\overline{c}, a\}}{\{\}}$$

If we would just remove complementary tiepoints, we would obtain in this last step the set CORNER$_6$.tps $= \{b, a\}$, which is counterintuitive. CORNER$_6$ covers the whole graph, so its not adjacent to any node and its tps should be empty therefore. But b and c are tiepoints of the overlapping component RA$_1$, and none of them is contained in the tps of both ANGLEs, so, following our tiepoint-inheritance rule, they have to be removed and we actually get an empty tps.

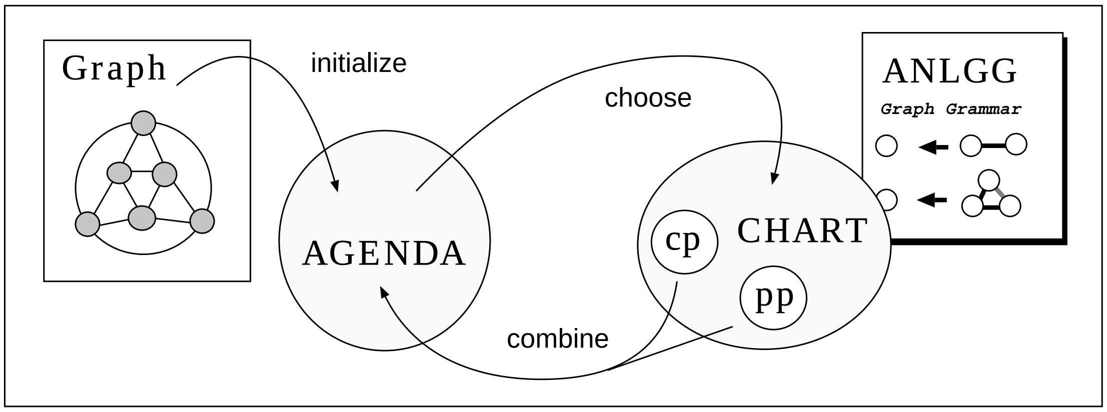

Figure 5: Architecture of the parser

`add-to-chart patch` adds a partial or complete patch to chart such that the access functions work as specified above. If they are implemented as tables a patch causes in general several entries, e.g. a cp is entered in CP once for each of its tiepoints and in CP$_{ov}$ once for each of its role values. Note that it is sufficient to enter a pp for the tiepoints of *one* of the already found neighbors of its next role, not for all, as mentioned in the specification of PP.

Introduction to chart parsing with string grammars may be found in [Ear 70, Win 83]. ANLGGs as presented here are similar to plex grammars, also known as hyperedge replacement systems. A parser for plex grammars, quite different from the one presented here, is given in [BuHa 89]. The parsing procedure presented here is an extension to the one introduced by R. Lutz in [Lut 89]. Main differences are the use of a search-guiding heuristic by ordering the agenda, the ability to specify overlappings, the ability to specify overlapping and neighborhood not only for nodes, but also for their components, and the use of sort hierarchies. Note also, that our total ordering of the productions right hand sides causes an early pruning and determines uniquely the order that the parser uses to build an instance, i.e. the instance is build once or never. In [Lut 89] an instance of a production whose rhs consists of n nodes will in the worst case be build in n! different ways leading to n! copies of the same patch.

4 Termination, Completeness, and Complexity

The agenda is initialized with one patch for each node of the input graph. The number of patches that can be build from that by binary combinations via `combine` is finite, if the grammar does not allow infinite chains of production applications. Because every application of `choose` removes one patch from the agenda and `combine` generates no patch twice, the agenda will run empty after a finite number of pairwise `choose-combine` applications.

After each such pairwise application the following chart invariant holds: For two arbitrary patches cp and pp in chart the direct successor i.e. binary combination of cp with pp is in agenda or in chart. When finally the agenda runs empty this means: every direct successor of any patch in chart is already in chart and by induction: every successor, that can be generated by a finite sequence of binary combinations of the input nodes is in chart, in particular all possible parses.

Our formalism is expressive enough to cope with graph isomorphy which is well known to be np-complete. This doesn't mean, that every grammar formulated within our framework has worse runtime. The strategy to avoid the combinatoric explosion inherent to the problem is here to incorporate domain-specific heuristics ($<_A$ and $<_p$) and sorts ($<_S$) to guide the parser's search.

5 Application to Feature Recognition in CIM

The parser has been implemented in Common Lisp and runs on SPARC stations with ivory boards. We developed graph grammars representing expert knowledge of our two experts in the turning and milling domain and tested the parser with several workpieces.

The milling grammar, still under development, contains 19 productions and a sort hierarchy with 43 sorts. Sort hierarchies proved to be very useful both for efficiency and readability of the grammar.

The workpiece in figure 6 is represented by a graph containing 50 nodes and 100 edges. The parsing took 2.12 seconds (CPU time) to derive a first of 48 possible parses. 1224 patches were generated. Further work is needed to improve the heuristics guiding the parsers search in order to be able to deal with more complex workpieces and to complete the graph grammar.

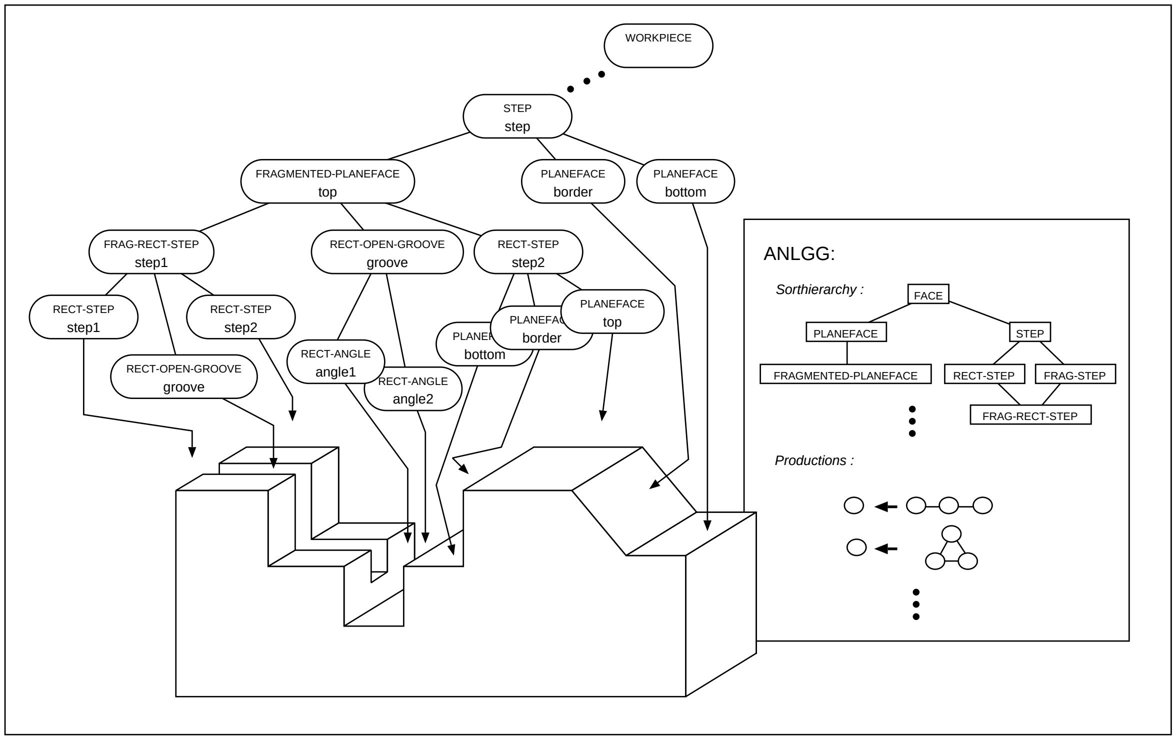

Figure 6: A workpiece and a part of the feature tree

The turning grammar contains 64 productions and a sort hierarchy with 27 sorts. One of the more complex shapes generates 587 patches to derive all 4 possible parses in 0.9 seconds (CPU time). In early days of the grammar the parser generated more than 2800 patches. The reduction to 587 was mainly caused by tuning the $<_p$ heuristic.

In our system PIM (Planning In Manufacturing, see [Leg 92]) the feature tree is used to guide the refinement and merging of skeletal plans associated with the sorts to get an overall plan for the manufacturing of the workpiece suitable for NC-machines.

To represent knowledge about intended solutions within PIM, a manufacturing expert associates with every feature a set of more or less detailed (fragments of) production plans, the so-called skeletal plans. A skeletal plan consists of a partly specified feature tree and constraints to trigger its application, and a partly specified plan, i.e. a list of actions.

The combination of features and associated skeletal plans represent the experience of the expert. It is important to realize that this observation implies that the application features and the skeletal plans depend on the concrete expert as well as on the concrete working environment and may be different for another expert or another environment.

The underlying AI-method of PIM is *heuristic classification*. Using the parser the abstraction of the workpiece is done by recognizing features and building a feature-tree describing the given workpiece. The association step is realized via simple links between features and skeletal plans. Using skeletal planning the refinement is done by merging skeletal plans and generating a complete process plan.

The idea of skeletal planning is straight forward: Find a first skeletal plan for the given goal (i.e a skeletal plan, whose tree matches the root of the feature tree). The plan contains some plan-actions which are subgoals to be reached in order to solve the whole problem. For every subgoal repeat the process until only elementary actions remain. The sequence of elementary steps is the intended plan.

6 Conclusion

In this paper we presented the 'made-to-measure' grammar formalism ANLGG and a heuristic driven chart based parser based on ANLGG for feature recognition in CIM. This tool supports the integration of any CA*-system via features ([Klau 92]) with any other CA*-system. The two possibilities to integrate heuristics as well as the use of a chart and sorts helps to find quickly a good first solution. This is necessary to support the communication between several CA* systems and to support the tasks of the systems ([Leg 92]).

The presented algorithm and formalism can also be used in other domains that have the same or similar characteristics (from the point of view of graph grammars) as the features in CIM.

References

[BGH 89] L.B. Booker, D.E. Goldberg, J.H. Holland: Classifier Systems and Genetic Algorithms, in: *Artificial Intelligence*, No. 40 (1989) 235-282.

[BuHa 89] H. Bunke, B. Haller: A parser for context free plex grammars, in: *M. Nagl (Ed.): Graph Theoretic Concepts in Computer Science, 15'th Workshop* (1989), 136-150.

[Cha 90a] T.C. Chang, Expert Process Planning for Manufacturing, *Addison-Wesley* (1990).

[Cha 90b] T.C. Chang et al, Feature extraction and feature based design approaches in the development of design interface for process planning, in: *Journal of Intelligent Manufacturing* (1990) 1-15.

[ChHe 91] S.-H. Chuang, M.R. Henderson, Compound Feature Recognition by Web Grammar Parsing, in: *Research in Engineering Design*, Springer-Verlag (1991) 147-158.

[Ear 70] J. Earley, An Efficient Context-Free Parsing Algorithm, in: *Communciations of the ACM* 13(2) (1970) 94-102.

[Ehr 79-90] H. Ehrig et al, Graph Grammars and Their Application to Computer Science, 1th - 4th International Workshop, *Springer Verlag*, LNCS 73, 153, 291, 532 (1979-1990).

[SuEr 90] C. Suttner, W. Ertel, Automatic Acquisition of Search Guiding Heuristics, in: *IJCAI'90* (1990) 470-484.

[FiSa 90] S. Finger, S. Safier, Parsing Features in Solid Geometric Models, in: *ECAI'90* (1990) 566-572.

[Klau 91] Ch. Klauck et al, FEAT-REP: Representing Features in CAD/CAM, in: *IV International Symposium on Artificial Intelligence: Applications in Informatics* (1991) 245-251.

[Klau 92] Ch. Klauck et al, Feature based Integration of CAD and CAPP, in: *CAD'92: Neue Konzepte zur Realisierung anwendungsorientierter CAD-Systeme*, Springer-Verlag, (1992) 295-311.

[Leg 92] R. Legleitner et al, PIM: Skeletal Plan based CAPP, in: *International Conference on Manufacturing Automation*, forthcoming (1992).

[Lut 89] R. Lutz, Chart Parsing of Flowgraphs, in: *IJCAI-89* (1989) 116-121.

[Mau 92] J. Mauss, Ein heuristisch gesteuerter Chart Parser für attributierte Graph Grammatiken, *Diplomarbeit*, Universität Kaiserslautern, (1992).

[Win 83] T. Winograd, Language as a Cognitive Process, Vol. I: Syntax, Addison-Weseley (1983) chart parsing: 116-127.