Towards a Parser for Mathematical Formula Recognition

Amar Raja, Matthew Rayner, Alan Sexton*, and Volker Sorge*

School of Computer Science, University of Birmingham, UK,
ug25ayr|ug98mxr|A.P.Sexton|V.Sorge@cs.bham.ac.uk
http://www.cs.bham.ac.uk/~aps|~vxs

Abstract. For the transfer of mathematical knowledge from paper to electronic form, the reliable automatic analysis and understanding of mathematical texts is crucial. A robust system for this task needs to combine low level character recognition with higher level structural analysis of mathematical formulas. We present progress towards this goal by extending a database-driven optical character recognition system for mathematics with two high level analysis features. One extends and enhances the traditional approach of projection profile cutting. The second aims at integrating the recognition process with graph grammar rewriting by giving support to the interactive construction and validation of grammar rules. Both approaches can be successfully employed to enhance the capabilities of our system to recognise and reconstruct compound mathematical expressions.

1 Introduction

Automatic document analysis of mathematical texts is highly desirable to further the electronic distribution of their content. Having more mathematical texts, especially the large back catalogues of mathematical journals, available in rich electronic form could greatly ease the dissemination and retrieval of mathematical knowledge. To build a robust system with high accuracy in correctly analysing mathematical texts, it is necessary to combine an effective optical character recognition (OCR) system with higher level syntactic and semantic analysis. However, while this is fairly routine for ordinary document analysis, when dealing with mathematics the particularities of mathematical notations and the often two dimensional nature of mathematical expressions have to be taken into account. Therefore, to date there are only very few systems available to integrate both processes. (The *Infty* system is a notable exception [6, 11].)

In [10] we presented a novel approach to OCR for scientific and mathematical texts. It is based on a large database of glyphs[1] together with a recognition algorithm that employs features computed from recursive geometric moment invariants [1, 4]. The approach is well suited for recognising subtle differences in

* The authors' work was supported by EPSRC grant EP/D036925/1.

[1] A glyph is a single, connected, shape of pixels. Characters are often composed of more than one glyph, e.g. "j" contains two glyphs and "≡" contains three.

characters such as different fonts and sizes, which are especially important in a mathematical context. The result of the recognition process is, for each recognised glyph, its best match from the database together with a LaTeX command that produces the glyph. The commands can then be used to reproduce an approximation of the input document by placing them at the original position of the recognised glyphs. We are currently extending this recogniser with higher level features that will enable both a more informed recognition process by generating feedback for the OCR as well as more advanced document analysis by recognising compound mathematical formulas and translating them into proper LaTeX expressions.

In this paper we present two higher level features we have recently integrated into our OCR system. The first feature, presented in Sec. 3, uses a well known technique from the document analysis literature, Projection Profile Cutting (PPC) [7, 12], and employs it in an innovative way. While the technique is traditionally used as a preprocessing step to segment mathematical formulas before an OCR step, we use it as a postprocessing step to our OCR system in order to reassemble the original structure of more complex mathematical expressions. This has three advantages:

1. With the information on size and position of glyphs in an expression, gained from the character recognition step, we can simply *compute* profile cuts rather then search for them as in the original technique.
2. Our recogniser is explicitly designed to be a glyph recogniser rather than a character recogniser. This is because problems of character segmentation and problems of character layout and decoration are often difficult to distinguish. The former are usually dealt with in a character recognition phase and the latter in a structural analysis phase. By using a glyph recogniser, these two, often conflicting, issues can be dealt with together during structural analysis. Thus we use PPC to treat character reconstruction from glyphs as just more structural analysis of the same form as reconstructing entire formulas.
3. The knowledge of the recognised glyphs and their original position gives us a uniform handle to overcome the old problem of PPC, namely that it cannot deal in a uniform way with enclosed characters.

The second higher level feature, described in Sec. 4, explores graph grammar rewriting [2, 5] approaches to the structural analysis of mathematical formulas. A graph is constructed where the nodes are the recognised glyphs of the formula and the edges record the spatial relationships between the nodes. A set of graph rewriting rules record subgraph patterns which, when matched, can be rewritten to non-terminal nodes which record the recognised formula sub-expression. The resulting graph can be further rewritten until, finally, the graph consists of a single node containing the fully recognised formula.

Such an approach depends critically on having a database of rewrite rules that describe the graphical grammar of mathematical formulas. This database is unlikely ever to be complete, given not only the huge range of mathematical conventions currently in use but also mathematicians' propensity to invent new

ones. Furthermore developing graph grammar rules is notoriously subtle and error-prone. Therefore, there is a need to be able to quickly and easily create new rules and to visualise the graph rewriting process of rule sets on actual graphs of mathematical formulas. We describe our first steps in developing a tool to interactively and easily create such rules from an analysis of images of formulas and to explore the operation of graph rewriting with these rules.

2 Database-driven Mathematical OCR

In [10] we have presented a database-driven approach to mathematical OCR by integrating a recogniser with a large database of LaTeX symbols in order to analyse images of mathematical texts and to reassemble them as LaTeX documents. The recogniser itself is based on a novel application of geometric moments that is particularly sensitive to subtle but often crucial differences in font faces while still providing good general recognition of symbols that are similar to, but not exactly the same as, some element in the database. The moment functions themselves are standard, but rather than being applied just to a whole glyph or to tiles in a grid decomposition of a glyph, they are computed in every stage of a recursive binary decomposition of the glyph. All values computed at each level of the decomposition are retained in the feature vector. The result is that the feature vector contains a spectrum of features from global but indistinct at the high levels of the decomposition to local but precise at the lower levels. This provides robustness to distortion because of the contribution of the high level features, but good discrimination power from those of the low levels.

Since the recogniser matches glyphs by computing metric distances to given templates, a database of symbols is required to provide them. We have developed a large database of symbols, which has been extracted from a specially fabricated document containing approximately 5300 different mathematical and textual characters. This document is originally based on [8] and has been extended to cover all mathematical and textual alphabets and characters currently freely available in LaTeX. It enumerates all the symbols and homogenises their relative positions and sizes with the help of horizontal and vertical calibrators. The single symbols are then extracted by recognising all the glyphs that a symbol consists of, as well as their relative positions to each other and to the calibrators. Each entry in the database thus consists of a collection of one or more glyphs together with the relative positions and the code for the actual LaTeX symbol they comprise. The basic database of symbols is augmented with the precomputed feature vectors employed by the recogniser.

The recogniser returns all the glyphs it encounters on the input document and for each glyph, a sequence of alternative characters in diminishing order of quality of visual match. In addition it provides information on the coordinates of the original glyph in the input document and on the size of this glyph by specifying its bounding box. The latter information is particularly useful in determining the actual size of a character in question in order to find the matching font size or scaling factor for the database match. The former information is exploited

for reproducing the input document. As previously we had no semantic analysis or syntactic parsing of the results to provide feedback or context information to assist the recognition, we could only reconstruct a document by composing a LaTeX picture environment that explicitly places, for each recognised character, the appropriate LaTeX command in its correct location.

3 Projection Profile Cutting

Projection profile cutting (PPC) is a technique that is widely used in different areas of document analysis. For the analysis of mathematical expressions it was introduced by Okamoto *et al* [7] for the case of printed mathematics and by Faure and Wang [12] for the case of handwritten mathematics. Other work that uses related techniques is, for instance, reported in [3]. All these projects have in common that they apply the PPC as a preprocessing step for the actual character recognition step to gain information on the formula structure and thereby to ease the recognition as well as, possibly, to correct symbols. In our case the aims and sequence of operations are rather different: We have already performed the glyph recognition of a mathematical text or expression. Thus we have, for each glyph in the text, its position on the page and the size of its bounding box together with a priority list of glyphs from our database that has been computed as the best matches by the recogniser. We now want to use PPC (1) to support the correct recognition and reassembling of multiglyph characters, and (2) to recursively assemble single characters to larger compound expressions that eventually can be expressed in a meaningful LaTeX command. One advantage of our approach is that we can deal uniformly and effectively with formulas that are both vertically and horizontally enclosed. These are generally inaccessible to traditional PPC techniques, where only some cases can be dealt with by including specialised rules. In the remainder of this section we will first introduce the technique in general and present the advantages of our approach by handling an expression containing a square root.

3.1 Basic Technique

The basic idea of PPC is to put straight projection lines in between components of an expression in order to separate the expression into elements that do not overlap. We start first with a vertical projection and then perform, on the resulting components, a horizontal projection. If no horizontal projection is possible we have obtained an *atomic component* of the expression, otherwise we proceed recursively until all the atomic components are found.

The fundamental element of PPC is the task of grouping the symbols by finding out which symbols overlap. We can define *vertical overlap* of two symbols as the situation where we cannot find a straight vertical projection that passes between the two symbols. Analogously, we can define *horizontal overlap* when we cannot perform a horizontal projection. To illustrate this consider Fig. 1: A symbol is essentially given by the height and width of its bounding box,

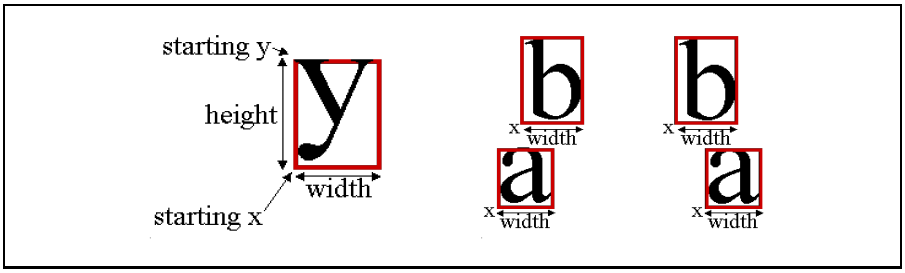
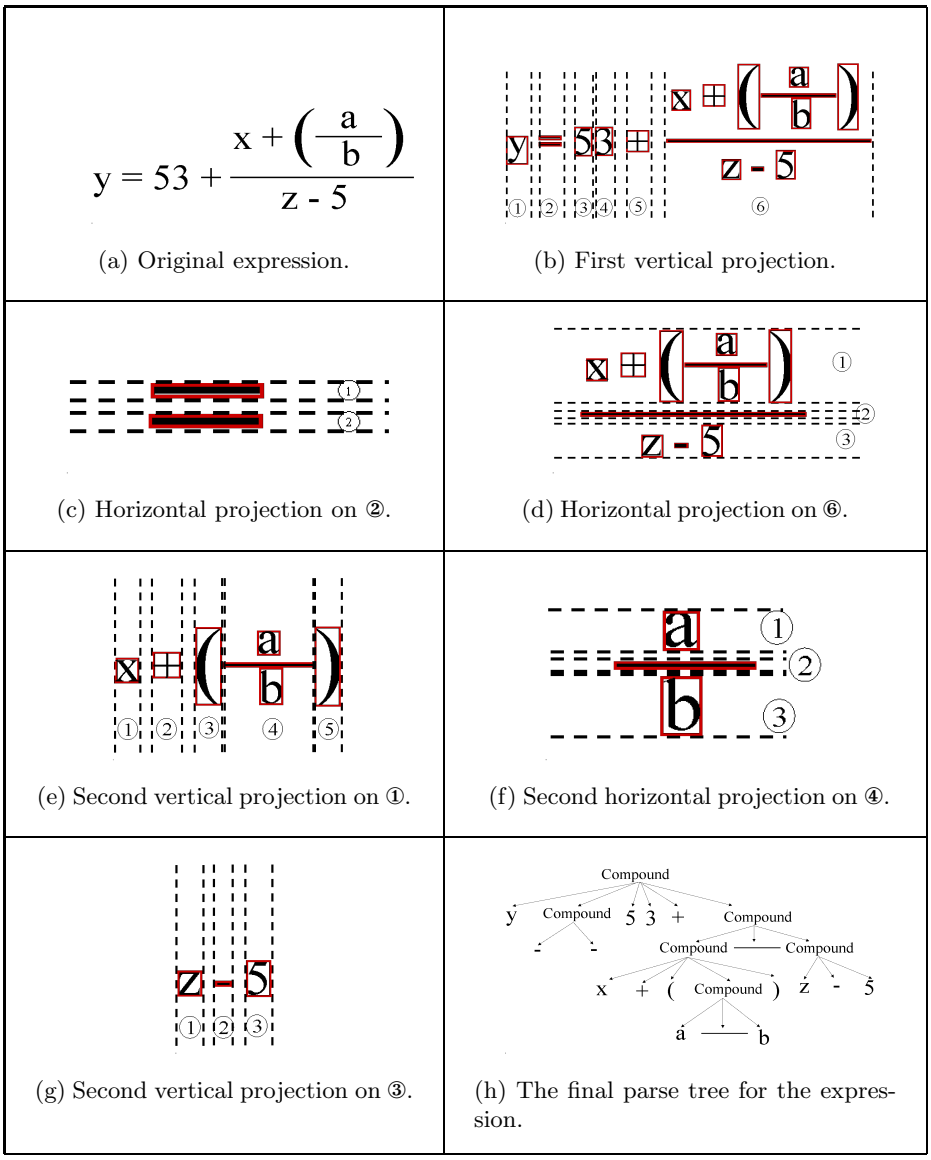


Fig. 1. Bounding box coordinates and overlap of symbols.

as displayed by the character 'y' on the left hand side. Now if the characters are arranged in a way that the bounding boxes overlap on a vertical line or a horizontal line one cannot perform the respective projection. The two possible cases of vertical overlap are, for instance, given on the right hand side in the figure.

Two symbols don't need to overlap directly for them to be classed as overlapping, they can overlap indirectly via a third symbol. Consider, for example, the expression in Fig. 2(a). The 'a' and the 'z' do not overlap but because the 'a' overlaps with the large divide sign and the 'z' also overlaps with the large divide sign then the 'a' and 'z' are classed as overlapping and are therefore grouped together in a sub-expression. To successfully group the symbols in the right subexpressions the order in which the symbols are checked for overlap is vital. We therefore sort the symbols in descending order by their width for vertical projections, and by their height for horizontal projections. This would mean the first two symbols to be checked for overlap would be the two divide signs for vertical projection. Since they indeed overlap, they will be grouped in the same subexpression together with all other symbols that overlap with them. Observe that this approach cuts out the search for the projection lines that is necessary in the traditional approach when segmentation is applied before the recognition process. In our approach we can simply compute overlaps using the bounding box information of each glyph and thus group the characters without search.

The entire PPC for the expression in Fig. 2(a) is given in the remainder of Fig. 2. Since we have the data from the OCR of the expression we already know exactly the position and sizes of the bounding boxes for the single glyphs, which simplifies the projection phase considerably. The first vertical projection, given in 2(b), then yields six different components. For components ① and ③–⑤ no horizontal projections are possible and we have atomic components. In fact, it is not necessary to test this explicitly since we can infer this already from the OCR data. Component ② on the other hand can be split again by the horizontal projection given in 2(c) to yield two atomic components. For the fraction in component ⑥ we need several recursive projection steps to fully analyse the expression. The first horizontal projection results in three components where only the fraction bar is atomic. While the denominator can be fully decomposed in another vertical projection (2(g)), the vertical projection on the enumerator



(a) Original expression.

(b) First vertical projection.

(c) Horizontal projection on ②.

(d) Horizontal projection on ⑥.

(e) Second vertical projection on ①.

(f) Second horizontal projection on ④.

(g) Second vertical projection on ③.

(h) The final parse tree for the expression.

Fig. 2. Recursive PPC example.

(2(e)) yields another embedded fraction as non-atomic component. This one can finally be decomposed in another horizontal projection (2(f)).

The result of the PPC is a parse tree that details the subcomponent relationship of the single glyphs in the expression (see Fig. 2(h)) This tree can now be used to assemble the resulting LaTeX expression that reproduces the original

input expression. It is worth noting that fraction bars in the expression can be determined as the LaTeX `\frac` command with the help of the OCR output. The resulting LaTeX expression is then of the form:

```
y = 53 + \frac{x + (\frac{a}{b})}{z - 5}
```

Observe that for clarity and to preserve space, we have omitted a `\mathrm` argument that actually embeds each of the alpha characters. Another noteworthy character in the above expression is the equality sign, whose reconstruction is necessary from the glyphs identified by the recognition engine. The next section explains how the combined '=' is obtained.

3.2 Recognising Multi-glyph Characters

During the OCR process each glyph is recognised separately and matched against the glyphs in the database. This yields a priority list of matching glyphs, with the best matches coming first. Thus, for reasons of noise, distortions due to scaling, artifacts of the scanning process etc., a glyph belonging to a character that is composed of multiple glyphs, might not have, as its identified closest match in the glyph database, the corresponding glyph of the appropriate character. For instance, the best match for a single bar of the equality sign could be a minus sign or the '$\cdot$' of a character 'j' might be best matched with dots from any one of multiple other symbols.

We can now exploit the results of the PPC to choose from the list of best matches for each glyph returned by the recogniser and reconstruct the correct character. When we find several atomic components in a single subexpression that are not separated by any non-atomic subexpression and that are within a certain distance threshold, the program parsing the expression tree attempts to find a single multi-glyph character that might match exactly with the components. It thereby considers a fixed number of multi-glyph characters it finds in the list of best matches given by the OCR program.

For example, for the equality sign it would take the two horizontal bars of the subexpression and find, in the matches list, the symbol '=', which indeed contains two bars with the right gap. For the character 'j', the OCR system returns '$\cdot$' and '$\jmath$' (i.e., the LaTeX commands `\cdot` and `\jmath`) as best matches. But since the '$\cdot$' of the 'j' overlaps vertically with the lower part of the 'j' they are grouped together in a vertical projection, which allows the program to identify the complete symbol 'j' which is in the list of best matches.

3.3 Handling Enclosed Expressions

PPC is a very effective tool for determining the spatial relationships between symbols in mathematical expressions. However, the method fails if symbols in an expression are both horizontally and vertically enclosed. Then neither vertical nor horizontal projection can penetrate the expression to extract all components of the formula. The classical example of a mathematical symbol that

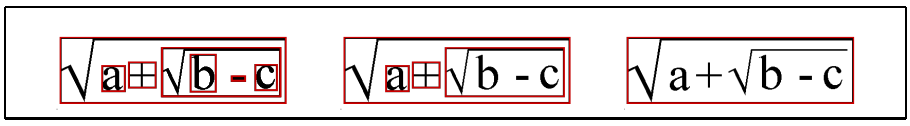


Fig. 3. Descending and assembling an expression with partially enclosed terms.

is impenetrable is the root symbol. However, there are various other example of symbols that fence expressions from several sides, or even contain them entirely, in Mathematics and, particularly, in Logics. For our discussion consider the following expression:

$$\sqrt{\mathrm{a} + \sqrt{\mathrm{b} + \mathrm{c}}}$$

Obviously, first vertical and then horizontal projection will yield no decomposition at all. However, from the information given by the OCR, we know that the expression consists of several non-connected glyphs and thus the projections should yield several atomic components. The program now further exploits the given information on the bounding boxes of the single glyphs to realise that there are indeed some glyphs fully contained in the bounding boxes of other glyphs. The leftmost expression in Fig. 3 shows the situation that presents itself to the program at this point. Before carrying on with the ordinary PPC, the algorithm first descends into the expression and extracts those components that are contained inside another glyph's bounding box. Thus in a first step it would extract $\mathrm{a} + \sqrt{\mathrm{b} + \mathrm{c}}$ from within the outer root symbol and in a second step $\mathrm{b} + \mathrm{c}$ from the inner root symbol. For this expression the regular projection algorithm will take over again and disassemble it into its components. Once the innermost atomic components have been been found the program will recombine those with the enclosing expression, that is the root symbol, and carry on doing so until it reaches again the top layer and the entire expression is analysed. The single steps of this process are depicted in Fig. 3. With the component tree constructed during the entire projection, we can then assemble the corresponding LaTeX command `\sqrt{a+\sqrt{b+c}}`. Again we have omitted the `\mathrm{}` around the 'a', 'b' and 'c'.

4 Graph Grammar Rewriting

Graph grammar rewriting [2, 5] is a very powerful technique that generalises string rewriting from standard string parsing to a graph parsing model. Rules specify a (possibly terminal) graph fragment to match and a non-terminal graph fragment to replace it with. Most such systems, when used for mathematical formula recognition, restrict the replacement fragment to be a simple node.

There are two major differences between our starting position and those of other projects using this technique. First, instead of using a character recogniser, we are using a glyph recogniser as discussed in Sec 2. Thus, part of our aim is to

reconstruct characters from glyphs in the rewriter. The point of this is that the perennial problem of segmenting characters, which are sometimes broken, sometimes touching other characters, can often only be resolved with contextual, and sometimes semantic information. It is difficult to provide the necessary range of such information to a character recogniser, as that information is generally discovered by the structural analyser. Transferring responsibility for character assembly from the optical recogniser to the structural analyser provides a more modular structure that allows simpler interactions between the recogniser and the structural analyser. This in turn allows easier development of more sophisticated techniques for applying structural and contextual information to the identification of characters.

Second, our intention is to provide a test framework for exploring different rule sets, and, eventually, different types of graph rewriting, rather than, at this point, a definitive graph rewriting system and a definitive set of rules for mathematical formula recognition. The tool we built to do this provides assistance for matching graph grammar rules to graphs, identifying conflicting rules, choosing and applying a subset of those rules and visualising, graphically, the entire process. The idea is that providing a very convenient way to manually parse the formulas and graphically add new rules dynamically, provides an excellent environment in which to design and develop large and complex rule sets.

4.1 Constructing the graph

A critical factor in any graph grammar rewriting approach is how to build the initial graph. If too many edges are generated then the resulting graph matching and parsing complexity may be too high. Too few edges mean that important connections between glyphs are overlooked and the formulas cannot be recognised. Lavirotte and Pottier construct their graph using compass point directions from each character [5]. While they report good success with this decision, our situation where we also need to reconstruct characters from glyphs may require finer distinctions. Also they do not report that they can handle enclosing constructs such as square roots.

Our graph is created initially with the best matching glyphs from the database for the image being analysed as the nodes. We could generate the complete graph in order to construct the edges but that would add considerably to the computational cost of the graph submatching algorithm. Also, it would not, we believe, add practical matching options to the system as the rewrite rules tend to work locally in the graph: neighbouring sets of nodes are rewritten into single nodes in a bottom up fashion. Hence connections between nodes corresponding to glyphs that are spatially distant in the image are unlikely to be useful. Therefore we chose to build our edges on a *line of sight* basis. Thus we create an edge between the nodes for two glyphs if there is a direct line between a pixel of one and a pixel of the other that is not obscured by any other glyph. In our current implementation, for performance reasons, we simplify that to a line from the bounding box centre of one to any point in the bounding box of the other. The edges are annotated with the centre point distances and relative directions

which can be easily generalised to fuzzy distances and directions. This approach supports enclosures in a direct and simple way.

4.2 Rules

The rewrite rules identify a *principal* node (or principal for brevity) around which the rewrite will take place. Each rule has a name. Terminal glyph nodes are named based on their glyph identifier, non-terminal nodes are named by the rules that created them. Rules contain constraint information about relationships between neighbouring nodes in the graph that must exist before the rule can be triggered. A number of attributes can be set for each connection required by a rule and are used to control the conditions that must be met for the rule to trigger. The system currently supports the following rule connection attributes:

Higher Top of the destination, but not the bottom, is higher than the principal.
Lower Bottom of the destination, but not the top, is lower than the principal.
TopLower Top of the destination is lower than the top of the principal.
BotLower Bottom of the destination is lower than the bottom of the principal.
Above Bottom of the destination is above the top of the principal.
Below Top of the destination is lower than the bottom of the principal.
Left Right edge of the destination is left of the left edge of the principal.
Right Left edge of the destination is right of the right edge of the principal.
PartLeft Destination is not left, but left edge is more left than the left edge of the principal.
PartRight Destination is not right, but right edge is more right than the right edge of the principal.

All these attributes can be set to values indicating that the specified condition must be met by this connection, must not be met or can be ignored. An extra `tag` attribute is used to identify the nodes that must be found at the end of the connection. This can be an individual node name, a list of possible glyphs or a pattern of node names. Other than the qualitative constraints provided by the above attributes, quantitative constraints can also be specified controlling relative size of the principal and destination nodes (based on height, width or bounding box area) and length of the connection relative to the dimensions of the principal.

4.3 Subgraph matcher

Our current matcher is simple and intended to provide us with a working test framework within which to explore graph grammar rule sets, visualisation of graph parsing and rule extraction tools before we proceed to a more sophisticated matching algorithm such as that of Rekers and Schürr [9].

The matcher takes the set of rules and systematically applies them to every glyph within the formula, taking the current glyph as the principal. All connections are first filtered by the destination tag and then all resulting combinations are checked. On a successful match the matching glyphs are placed in a collection ready for further analysis. In the event that a principal finds more than one

object which match the results of a certain definition, all matches are returned. For example, for the formula $w' \in \mathcal{W}$, we could have, for a naïve implementation of the rule handling set membership, the following potential matches returned: "w element in set $\mathcal{W}$", and "Prime element in set $\mathcal{W}$". These, of course, conflict with the appropriate rule in this case which would also be returned; namely that for matching the "w'". They conflict because every pair of these three rules have at least one node in common that they consume in order to rewrite the subgraph. Of course, only a rewrite of the w' rule will eventually result in a successful parse of the expression.

The ability to identify and explore sets of rules that conflict is an essential aspect of the design, debugging and development of rule sets. We say that two rules *conflict* if the intersection of the set of objects that they match is non-empty. A *conflicting rule set* is defined inductively as follows:

1. If rules r, r' conflict then $\{r, r'\}$ is a conflicting rule set.
2. If C is a conflicting rule set and r is a rule that conflicts with a rule $r' \in C$, then $\{r\} \cup C$ is a conflicting rule set.

A *conflicting rule set* $C \subseteq A$ is *maximal* in A if every $r \in A \setminus C$ does not conflict with any element of C.

Once the matcher has run on all rules, it partitions the set of matching rules into a set of maximal conflicting rule sets. All single element rule sets can be safely applied without conflicting with any other rule in the set of matching rules and are collected together in a *non-conflict group* of rules. Each other maximal conflicting rule set is recorded as a *conflict group*.

Once this phase is complete, the matcher attempts to resolve all the conflicts by finding a sequence of rule applications, one from each of the conflict groups, which will terminate with the least number of un-consumed vertices. This results in the most complete parse compatible with the rule set and is executed by applying each rule in turn and then recursively running the match process again until no further rewrites or matches are possible. In the event that multiple sequences may lead to the same resulting formula, the rule with the highest precedence within the rule set is identified.

The matcher is embedded in a rule visualisation and exploration tool called the *Formula Reader*. An example of this tool running on an expression is shown in Fig. 4. The main pane shows the formula under analysis at a point part way through an analysis. At this stage many of the lowest level rules have been applied and the graph has been rewritten to a significantly smaller number of non-terminals and remaining terminal nodes. Bounding boxes are drawn around each node as well as the sub-nodes that have previously been consumed. Edges are drawn between nodes that have not yet been consumed in the rewriting. All edges and nodes are colour coded to help the user identify appropriate parts of the formula. The current set of nodes in the graph is shown in the tree list in the upper left. One can drill down to see the internal structure of the node. Clicking on a node in the tree list highlights the corresponding node(s) in the main pane and vice versa. The small pane to the bottom left lists the current groups of non-conflicting or conflicting rules (in this case there is only one such

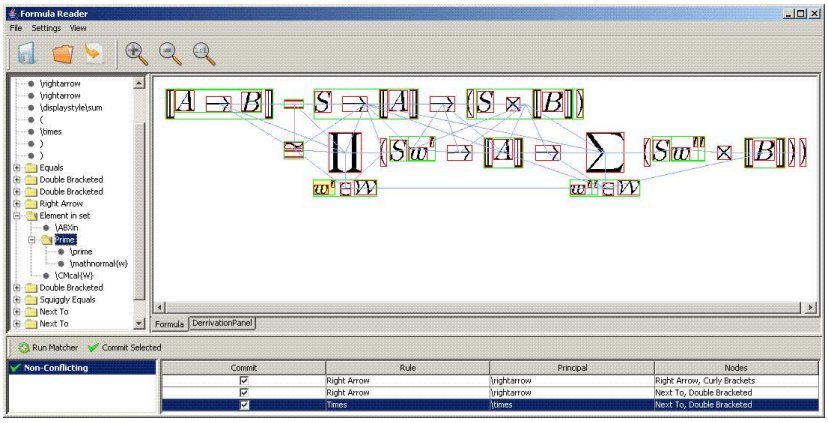


Fig. 4. Formula reader with the rule matcher in operation

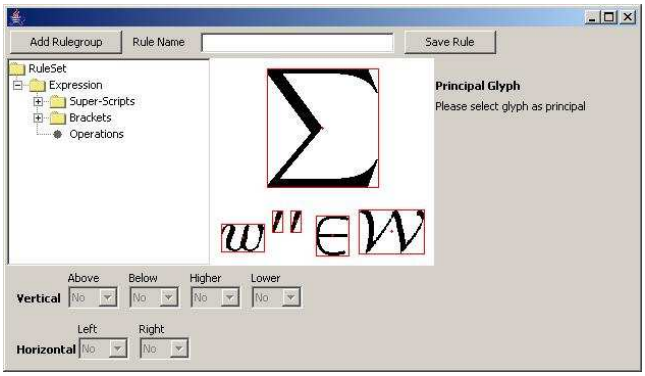


Fig. 5. Rule builder in operation

group, which is a non-conflicting one). To the bottom right the set of rules in the group is shown and the user can choose which of them to apply. By selecting an individual rule, the corresponding nodes and edges in the main pane are highlighted to help visualise the current state of the process. Committing the choices will apply the rules and set the system up to run the next round of the matcher. The whole system allows quick and easy visualisation and testing of individual rules, interactions between rules and operation of entire rule sets.

Rules can be created and added to a rule set using the rule builder facility. Fig. 5 shows an example. When working on a formula, a user can select a group of nodes from which to construct a new rule. A *principal* node must be chosen and then rules for each connected node can be added specifying the appropriate relationships for each one. The ability to add new rules dynamically during the process of a manual parse of an equation significantly reduces the difficulties of constructing large and complex rule sets.

5 Conclusion

We have presented an extension to our database-driven mathematical OCR system by adding two higher-level analysis features.

First we have combined the well-known projection profile cutting method with our approach to OCR and improved the ability of our recogniser to find proper matching multiglyph characters and added the ability to compose complex mathematical expressions as compound LaTeX commands. But this work has also yielded a new and robust solution to one of the major flaws of the PPC technique: that of penetrating enclosed expressions. This new approach works uniformly for all enclosing symbols and does not rely on special cases for certain symbols.

Second we have developed a first version of an interactive graphical tool that significantly assists in the visualisation, analysis and development of graph grammar rule sets — one of the major barriers to the study and use of this powerful technology in mathematical formula recognition. This will enable us to develop graph grammar based approaches to the structural analysis of mathematical texts.

References

1. F. L. Alt. Digital pattern recognition by moments. *Journal of the ACM*, 9(2):240–258, April 1962.
2. A. Grbavec and D. Blostein. Mathematics recognition using graph rewriting. In *Proc. of ICDAR'95*, pages 417–421, 1995.
3. J. Ha, R. M. Haralick, and I. T. Philips. Understanding mathematical expressions from document images. In *Proc. of ICDAR'95*, pages 956–959, 1995.
4. M.-K. Hu. Visual pattern recognition by moment invariants. *IEEE Transactions on Information Theory*, 8(2):179–187, Feb 1962.
5. S. Lavirotte and L. Pottier. Optical formula recognition. In *Proc. of ICDAR'97*, pages 357–361, 1997.
6. M.Suzuki, F.Tamari, R.Fukuda, S.Uchida, and T.Kanahori. Infty — an integrated OCR system for mathematical documents. In *Proceedings of ACM Symposium on Document Engineering*, pages 95–104, 2003.
7. N. Okamoto and B. Miao. Recognition of mathematical expressions by using the layout structures of symbols. In *Proc. of ICDAR'91*, pages 242–250, 1991.
8. S. Parkin. The comprehensive latex symbol list. Technical report, CTAN, 2003. available at www.ctan.org.
9. J. Rekers and A. Schürr. Defining and parsing visual languages with layered graph grammars. *J. Visual Languages and Computing*, 8(1):27–55, 1997.
10. A. Sexton and V. Sorge. A Database of Glyphs for OCR of Mathematical Documents. In *Proc. of MKM'05*, volume 3863 of *LNCS*, pages 203–216. Springer, 2006.
11. M. Suzuki, S. Uchida, and A. Nomura. A ground-truthed mathematical character and symbol image database. Technical report, Kyushu University, 2004.
12. Z. Wang and C. Faure. Structural analysis of handwritten mathematical expressions. In *Proc. of Ninth Int. Conf. on Pattern Recognition*, 1988.