Ralph-Johan Back | Victor Bos | Johannes Eriksson

MathEdit: Tool Support for Structured Calculational Proofs

Turku Centre *for* Computer Science

TUCS Technical Report
No 854, December 2007

MathEdit: Tool Support for Structured Calculational Proofs

Ralph-Johan Back
Åbo Akademi University, Department of Information Technologies
Joukahaisenkatu 3-5A, 20520 Turku, Finland
backrj@abo.fi
Victor Bos
victor.bos@gmail.com
Johannes Eriksson
Åbo Akademi University, Department of Information Technologies
Joukahaisenkatu 3-5A, 20520 Turku, Finland
joheriks@abo.fi

TUCS Technical Report
No 854, December 2007

Abstract

The structured calculational proof format emphasizes structure and readability by presenting derivations as outlined sequences of term transformations. The Mathematical Derivation Editor (MathEdit) is an effort to develop tool support for this format. It is a text editor with built-in support for an extensible mathematical syntax and structured derivation notations. In this paper we overview and discuss the features of MathEdit and their implementation: how the editor parses and understands mathematical expressions, determines applicable rules, and how structured derivations are represented. We demonstrate use of MathEdit through example derivations from process algebra.

1 Introduction

A well-defined hierarchical structure for mathematical proofs is important both for the author and the readers of a proof [24]. Structure forces the author to think his ideas through thoroughly before committing them to paper, and provides the reader with a description of the proof that is easy to understand and that can be examined at different levels of detail. The *structured calculational proof* [3] format is a proof presentation format that combines the readability of calculational proofs with the structure of natural deduction.

However, creating and maintaining a structured layout of complex derivations can be cumbersome without dedicated tool support. This raises the need for an editing environment that aids the author by imposing a well-defined structure on documents. But such a tool should not be overly restrictive either; authors of mathematical documents frequently need to mix formal notation with blocks of prose-like text.

This paper presents MathEdit, a tool developed in an academic software factory. MathEdit supports editing and correctness checking of structured calculational proofs in a familiar text-editor like environment. The user can write a derivation by manually typing in each step, or by applying rules in a point-and-click manner. With the click of a button, all derivations can be checked for well-formedness. The mathematical syntax of MathEdit is extensible and there are no built-in restrictions on the domains in which MathEdit can be used.

Aside from the simple equation editors integrated into many word processors, there exists a number of high-quality tools for editing mathematics-intensive documents. They encompass a wide range of applications; some focusing mainly on structure and presentation [27, 11], other are full-blown computer algebra systems [29, 16]. Proof management tools on the other hand often target a specific domain; examples of tools geared towards the formal verification of programs are [6, 23, 15]. The goal of MathEdit is to provide a platform supporting general non-domain-specific mathematics combined with the structured calculational proof paradigm. While this makes MathEdit very general and applicable to a wide range of problems, it also inevitably makes it much less "polished" than specialized tools; we will touch on this aspect further on in the paper.

The remainder of this paper is organized as follows: Section 2 provides an introductory overview of the structured calculational proof format. In Section 3 we discuss the features needed from in an editor for such proofs. Section 4 provides an overview of our editor, while Sections 5 and 6 describe in more detail how the editor parses and understands mathematical expressions, determines applicable rules, and how structured derivations are represented. Section 7 provides concrete examples of using MathEdit. We end with a

short summary and some discussion on future work.

2 Structured Calculational Proofs

The structured calculational proof format for writing derivations is based on the concepts of natural deduction [20] and calculational proof [21], combining and extending them to provide a well-structured, outlined layout for proofs that is visually pleasing and allows hierarchical decomposition. The following example illustrates the structured calculational proof format:

$$
\begin{array}{ll}
 & A \wedge \llcorner B \wedge A \lrcorner \\
\equiv & \{ \text{ use the first conjunct to simplify the second } \} \\
 & \quad \bullet \langle A \rangle \\
 & \quad B \wedge A \\
 & \quad \equiv \quad \{ \text{ use the assumption to replace } A \text{ with } \mathsf{true} \ \} \\
 & \quad B \wedge \mathsf{true} \\
 & \quad \equiv \quad \{ \text{ propositional calculus } \} \\
 & \quad B \\
 & A \wedge \ulcorner B \urcorner \\
\equiv & \{ \text{ commutativity of } \wedge \ \} \\
 & B \wedge A
\end{array}
$$

A derivation consists of a number consecutive interleaved *term* and *comment* lines. The first term ($A \wedge B \wedge A$) is transformed in a series of reduction steps, each step being based on a rule indicated in the comment line. In cases where a step transforms a subexpression, corner carets indicate the redex before ($\llcorner$ and $\lrcorner$) as well as after ($\ulcorner$ and $\urcorner$) the transformation. The symbol in the beginning of a comment line describes the relation between two consecutive terms (here $\equiv$, since the goal of this proof is to show that equivalence holds between the first and the last terms.) However, proofs are not limited to symmetric relations—any transitive relation will do—and it is also possible to use different relations in the same derivation, but in that case one need to consider carefully their combined relation.

The example also shows the use of a hierarchical outline and a *subderivation* with *contextual information*. In proving $B \wedge A \equiv B$ a subproof with the assumption A (enclosed in angle brackets) is set up as an indented subderivation under the first comment line. This *focusing* on a subexpression is useful for avoiding repetition of (potentially long and complex) unchanged parts of an expression while still maintaining correctness, and its use has been formalized in *window inference rules* [25].

If a subderivation is used in a step to establish a non-symmetric relation, such as implication, the monotonicity properties of the expression being

transformed must be taken into account. For example, conjunction is monotonic in either of its arguments with respect to implication, which allows us to write the following derivation to prove $(A \wedge P) \Rightarrow (A \wedge Q)$:

$$
\begin{array}{ll}
 & A \wedge \llcorner P \lrcorner \\
\Rightarrow & \{ \text{ since } \wedge \text{ is monotonic in its right argument } \} \\
 & \quad \bullet P \\
\Rightarrow & \quad \{ \text{ hint why } P \Rightarrow Q \text{ } \} \\
 & \quad Q \\
 & A \wedge Q
\end{array}
$$

There are other connectives that are not monotonic with respect to implication; implication itself, for example, is monotonic in its right argument but anti-monotonic in its left. Some operators are neither monotonic nor anti-monotonic. Subderivations that transform the arguments of such an operator must preserve equivalence only.

Subderivations are also used to prove *hypotheses* that are side conditions for *conditional* rules. As an example, consider the following arithmetic simplification:

$$
\begin{array}{ll}
 & 10/10 \\
= & \{ \; x/x = 1 \text{ provided that } x \neq 0 \text{ with } x := 10 \; \} \\
 & \quad 10 \neq 0 \\
= & \quad \{ \text{ by comparison } \} \\
 & \quad \mathsf{true} \\
 & 1
\end{array}
$$

When the first step is derived the hypothesis $10 \neq 0$ is postulated, making it the responsibility of the proof author to prove this relation in a subderivation. This use of subproofs differs from that of focusing rules in that the resulting expression of the subderivation is not included in the main derivation. Instead it is a proof obligation of the derivation step that the divisor is non-zero.

3 Tool Support for Structured Derivations

We believe that a tool for writing mathematics should provide and enforce structure whenever it aids the user in his goal, and otherwise offer as much freedom as possible. We chose the standard graphical text editor or word processor as the basic application model for MathEdit. The large majority of users are in some way familiar with this style of application, and have a general intuition about the user interface. A *document* containing lines of text is edited by moving a *cursor*, or caret, around the document while

issuing *commands*. Commands can be simple, such as typing a single character, or more powerful, such as performing a string-based search and replace operation on the entire document.

Outlining editors, sometimes called outliners, are document editors that allow their users to edit text, and possibly other elements, in a hierarchy. In a line-based outliner, *outlined text* consists of many indented lines, following the rule that each line can be indented at most one level to the right from the previous line. A line followed by indented lines is called *parent* and the indented lines are called *children*. A line together with all its children and sub-children is called an *item*; *atomic* items have no children. *Composite* items, i.e. items which have at least one child in contrast to atomic items, can be collapsed and subsequently expanded to hide and show their children.

An outlining editor provides good overview of a document by allowing the user to hide details deep in the hierarchy and get a "bird's eye"-view of the document, while at the same time making it possible to quickly delve into the deeper branches of the hierarchy for details. Outliners are useful when reading and writing mathematical proofs for this very reason—a complex derivation often contains several rule applications with perhaps equally complex subderivations. These steps can be collapsed to hide details that are not interesting when one is attempting to understand the general idea of the proof, and as soon as more information about a particular step is needed, that step can be unfolded and the subderivation revealed. The author is thus free to add as much material as possible to a document without fear that the result becomes incomprehensible. The idea of navigating a proof in this way is known as *proof browsing* [22].

Authors of mathematical documents are very conscientious about the correctness of their writings. Yet according to Lamport [24], even proofs published in mathematical journals frequently contain errors. As writing is an iterative process, it is especially important that changes and additions do not invalidate prior work. Human beings are, however, notoriously bad at frequently checking "trivial" things, such as whether a declaration is valid in some specific language or if a rule application is still valid. But these tasks can be quickly carried out by computers, which is why a tool should provide as much checking as possible of user input, albeit in a unobtrusive way. Furthermore, whenever the environment (assumptions, rules, lemmas) changes, the tool should make it possible to run an automated check on all existing proofs to ensure that they are still valid.

One example of a mathematics-oriented text editor is Math$\int$pad [11]. It is a strongly syntax-directed editor in which templates called *stencils* are used to define the visual and logical structure of syntactic elements in a document. The editor supports selection and manipulation based on this structure: for example, a specific stencil can define a certain mathematical operator, and clicking on that operator in a document will then select the

whole subexpression. Furthermore, it features a number of commands for rewriting expressions, such as Reverse, Distribute and Factorise, but these work on a purely syntactic level.

GNU TexMacs [27] is an editor based on similar ideas as Math$\int$pad. In addition it supports a few computer algebra systems, thus making it possible to include the semantics of such systems into TexMacs. Translation layers and interprocess communication are used to access the functionality of these systems from within the TexMacs environment, making the integration quite shallow.

We chose not to make MathEdit strongly syntax-directed. While a mathematical expression, such as an algebraic formula, has a tree-like micro-structure, we have not seen pressing needs to create a structured editing environment for such micro-structures. In practice, it is often useful to temporarily "break" a structure while doing edits. In MathEdit, derivations and mathematical expressions are treated as delineated sections in a free-form text document that have been explicitly indicated by the user to conform to a specific language; if a section does not, the error is reported and the user can correct it. We do, however, appreciate immediate feedback and MathEdit therefore implements automatic parsing during editing and visualizes the abstract syntax trees of expressions.

However, for elements such as formulas and derivations, we want a well-defined syntax and a means for including semantics. Such semantics makes it possible to reduce formulas using rules of the mathematical formalism rather than on a purely syntactic basis. The mathematical language provided by MathEdit is extensible in order to enable users of the tool to work with different kinds of mathematics. A basic syntax and simple Boolean and arithmetic operators are provided, but users can add new types, operators and identifiers without reprogramming the tool itself. Users are also able to declare custom rules and use these seamlessly together with built-in rules in derivations; this feature is crucial for usability, since it will get extremely tedious to read and write derivations consisting only of steps based on a small number of pre-defined rules. Furthermore, in more demanding situations, where a user might want to significantly change the mathematical language, this is also possible by writing a new *mathematical profile* (further described in Section 5).

It is not a goal of MathEdit itself to typeset formulas—the TEX and LATEX systems are the standard tools used for this purpose, mainly due to the excellent quality of the documents they produce. MathEdit should thus be able to export documents for further LATEX processing and typesetting. However, our goal has not been to create a LATEX front-end, as several such applications already exist.

4 Overview of MathEdit

In addition to producing the tool itself, an important goal of the MathEdit project has been to evaluate new software engineering techniques in practice. This section provides a brief overview of the methods and technologies used in the development of MathEdit; we then refer the interested reader to a number of technical reports describing these methods in more detail. To give the reader an idea of MathEdit's capabilities, we also briefly describe the main features.

4.1 Development

The Mathematical Derivation Editor was developed in the *Gaudi Software Factory* [8], an experimental software factory in an academic setting that aims to be a testing ground for new software development methods in practice. More than ten projects have been carried out at Gaudi since its inception in 2001, such as a basic outlining editor [9] and a personal financial planner [7]. Results from these projects have indicated that it is indeed possible to produce software in a timely manner despite the limitations on some resources in a university environment, including lack of funds and dedicated personnel.

Programmers employed in Gaudi are computer science and computer engineering students guided by graduate students who function as coaches. Some projects developed in Gaudi, such as MathEdit, are also research tools while others are built mainly to study the development process. A professor or graduate student with a research interest in the product being developed typically acts as the customer for a project.

The Gaudi software process borrows many practices from *Extreme Programming (XP)* [13]. XP is an agile software development process that has become increasingly popular for high-risk, high-velocity projects. Its main goal is to mitigate some of the most common risks in software development, including delayed schedules, requirements changes, high defect rates and developer turnover. This is achieved through a number of practices, including but not limited to:

- keeping an on-site customer,
- unit testing,
- pair programming,
- continuous integration,
- shared code ownership.

These practices have been implemented in Gaudi and the results so far indicate an increased reliability of the produced software.

MathEdit is built in a layered fashion using the *Stepwise Feature Introduction (SFI)* design methodology [4]. In SFI software is built in layers, such that each layer implements a certain feature or set of closely related features. Software is built in an incremental fashion so that the bottom layer provides the most basic functionality, and subsequent layers add more advanced functionality. The layers are implemented as class hierarchies such that a new layer inherits all functionality of previous layers by subclassing existing classes, and adds new features by overriding methods and/or defining new methods. A detailed overview of how SFI was applied in MathEdit can be found in both [5] and the master's thesis [17].

The main programming language used for implementing MathEdit is *Python* [28], an open-source, interpreted, dynamic, and object-oriented language with a clean syntax. Python is also fully object-oriented and has been used to build very large software projects. While Python excels at ease of use and speed of development, it essentially achieves these advantages by trading off execution speed; Python programs are in general much slower than programs written in compiled languages. The major part of the source, about 46 000 lines, is in Python, while some 2 000 lines are C++. The initial plan was to write all code in Python; C++ was only used out of necessity in a small number of performance-critical areas.

4.2 The Product

A screenshot of the running application can be seen in Figure 1. MathEdit runs on both Windows and Linux/X platforms and can be freely downloaded under a GPL license from `http://mde.abo.fi` .

MathEdit implements the basic functionality of a text editor. The user works with *documents*, which are visible and manipulated through on-screen *views.* A document can be associated with several views, and several documents can be open at the same time. Each view has a cursor which can be moved around independently, and text selection is performed with either the keyboard or mouse. An unlimited *undo/redo* mechanism makes it possible to undo editing commands in order to correct mistakes. The clipboard interaction commands, *cut, copy* and *paste,* allows copying text data between applications. *Unicode* [1] is used for all its internal text processing, so that mathematical and other useful symbols (such as Greek letters) can be represented.

The depth of a line in the outlining hierarchy is indicated by indentation. Collapsible items, i.e. lines with visible children, show a minus sign in the sidebar; clicking on the sign collapses the item, hiding the children. Conversely, collapsed lines show a plus sign which can be clicked to re-expand

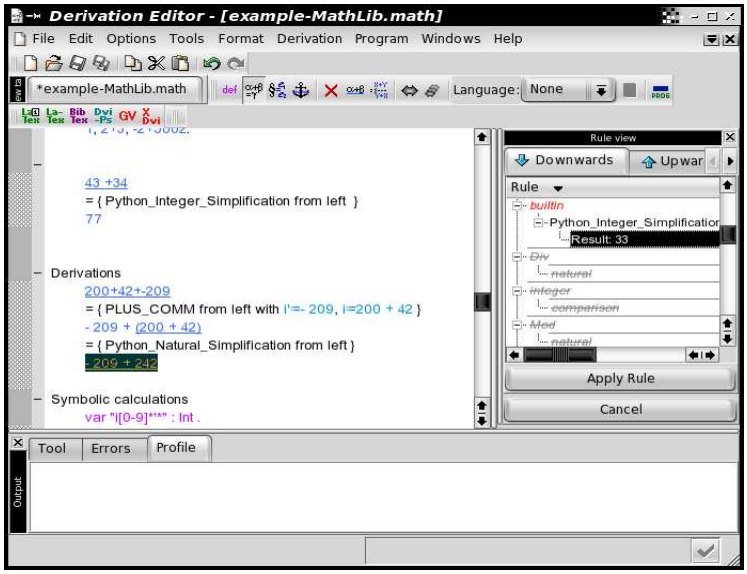

Figure 1: Screenshot

the line and show the children.

The user creates a mathematical expression by selecting a range of text and clicking a tool button or selecting a menu alternative. The selected text is then marked up in a special typeface and is considered a mathematical expression and becomes subject to automatic parsing. An accepted expression is shown in blue color, while an illegal expression is colored red.

Derivations are created by applying *rules* to mathematical expressions. Rules are typically Boolean expressions and can be provided by the mathematical profile (built-in rules) or entered by the user. To perform a reduction step the user selects a (sub)expression and clicks a button, whereby the editor shows all applicable rules. For each rule a preview of the reduction is shown, and the user can select which rule to apply. When a rule is applied, a new derivation step (possibly containing subderivations) is created and the transformed expression is inserted into the document.

For persistent storage of documents, MathEdit uses an XML file format conforming to a custom DTD. For presentation, documents can be exported to HTML and LaTeX formats. Both formats preserve the outlining structure and HTML files also implement the folding feature using dynamic HTML.

5 Mathematical Language

To make MathEdit a general tool for writing mathematics, an important feature of MathEdit is to allow the user to extend the mathematical language. The assumptions MathEdit makes about the mathematical language

are needed to provide support for structured derivations, see Section 6. These assumptions are defined in the *mathematical profile interface* (MPI): an API defined as a Python module. The main data structures defined in the MPI are *Expressions* and *Rules*. The main operations on these data structures are *parsing expressions*, *selecting subexpressions*, *applying rules*, *constructing derivations*, and *checking derivations*.

There are three ways to make extensions to the mathematical language of MathEdit. The most powerful way is to provide an implementation of the MPI. This means that the data structures and operations need to be implemented in Python. Therefore, these kind of extensions are for programmers only.

The second way to extend the mathematical language is to use the *Universal Profile* (UP*)*: an MPI implementation that allows allows the user to add new mathematical notations. UP itself is an example of an extension written in the first way. However, when using UP to extend the mathematical language, one is just writing in MathEdit and one does not need to have any programming experience.

The third way to extend the mathematical language of MathEdit is to combine the first two ways. This is useful if the extension writer wants to provide "built-in" rules which manipulate expressions in more efficient ways than the standard UP built-in rules. For example, there is an extension that uses the HOL theorem checker to check (parts of) structured derivations, see [18]. The mathematical language of this extension is defined in an ordinary MathEdit document using UP. The communication between MathEdit and HOL is written as an MPI implementation in Python.

5.1 The Universal Profile

A good way to learn about UP is to open a new document in MathEdit and select the UP profile for this document (use the *File*▷ *Use Profile*▷ *UP* menu). Before typing anything at all, list the grammar of the profile (menu *Derivation*▷ *Show Grammar*). In the *Profile*-tab at the bottom of the MathEdit window, the grammar and some additional information about everything that is predefined in UP will be displayed.

UP defines, among others, the following types: *Term*, *Bool*, and *Identifier*. The following subtype relations hold:

$$Bool \subseteq Term \tag{1}$$
$$Identifier \subseteq Term \tag{2}$$

There are two atomic (indivisible) *Bool* expressions: $\top$(true) and $\bot$(false). Instead of using these special symbols, one can use `_T_` and `_F_`, respectively.

Although these expressions are defined internally by UP, they behave as if they were defined as follows.

```
op TRUE: "⊤|_T_" → Bool .
op FALSE: "⊥|_F_" → Bool .[1]
```

Note that MathEdit does not make a distinction between constants and operators; constants are just nullary operators, i.e., argument-less operators. The identifiers `TRUE` and `FALSE` are the names of the expressions. Following the name is a colon and, in this case, a double-quoted regular expression. The regular expression should be written according to the syntax of Python's regular expressions. `"⊤|_T_"` is the double quoted regular expression that matches either one ⊤ symbol, or the three character string `_T_`. Similarly, `"⊥|_F_"` is the double quoted regular expression that matches either one ⊥ symbol, or the three character string `_F_`. The result type of these operators is *Bool*.

There is one operator in UP that defines *Identifiers*. It is defined as follows:

```
op Identifier: "[a-zA-Z][a-zA-Z0-9_']*" → Identifier .
```

The regular expression matches strings starting with a lower case or upper case letter and ending in zero or more letters, digits, underscores, or single-quotes. Examples of such identifiers are `a`, `B`, `a_B`, `a10`, `a'`, `a''_b`, and `a_''10a`.

In addition to these operators, or constants, UP defines an equality and an inequality operator for each type. For instance, for the types `Term`, `Bool`, and `Identifier` it defines the following operators.

```
op EQUAL_Term:
    Term "≡|=" Term → Bool [prec=0] .
op NOT_EQUAL_Term:
    Term "≢|≠" Term → Bool [prec=0] .
op EQUAL_Identifier:
    Identifier "≡|=" Identifier → Bool [prec=0] .
op NOT_EQUAL_Identifier:
    Identifier "≢|≠" Identifier → Bool [prec=0] .
op IDENTICAL_Bool:
    Bool "≡|=" Bool → Bool [prec=0] .
op NOT_IDENTICAL_Bool:
    Bool "≢|≠" Bool → Bool [prec=0] .
```

[1] The `prec` attribute of the operators is not shown here, because both `TRUE` and `FALSE` have the default value (10) for this attribute.

The operators `EQUAL_Term`, `EQUAL_Identifier`, and `IDENTICAL_Bool`, are interpreted as syntactic equality.[2] However, the user is free to define more rules for any of these operators, thereby making the equality operators less strict.

UP does not define any rules for the negated operators `NOT_EQUAL_Term`, `NOT_EQUAL_Identifier`, and `NOT_IDENTICAL_Bool`. However, usually the user will want to define them as the negated form of the equality operators. This is possible by defining a rule of the form

```
rule NOT_EQUAL_Term: (t ≢ s) ≡¬(t ≡ s) .
```

Here, `s` and `t` are variables of type `Term`. Note that to define such rules, Boolean negation, ¬, is required. This operator is not predefined in UP.

6 MathEdit Support for Structured Derivations

In this section we consider the various syntactical elements that make up structured derivations. We define a syntax for derivations; the goal being a machine-readable syntax maintaining the clarity of the structured calculational proof format. We then discuss how MathEdit unifies terms and rules, and finally we present the strategy used to check derivations.

6.1 Elements of Structured Derivations

6.1.1 Term lines

Each term line consists of a term, and a redex indicator (except for the last term, which is not further reduced). In contrast to the notation in [3], MathEdit marks redexes with underlining rather than corner carets, and does not mark the result of the reduction in the sequel term. Also, in MathEdit it is possible to reduce a subexpression without introducing a subderivation, which results in duplication of the unchanged parts of the term. This feature is mainly intended to be used on small terms, since duplication can make proof maintenance unwieldy; in such cases introducing a subderivation (using, e.g., a focusing rule) is recommended.

6.1.2 Comment lines

The relation symbol in the beginning of each comment line describes the mathematical relation between the term immediately before and the term

[2] The reason `IDENTICAL_Bool` has a different kind of name than the `EQUAL_`*Type* operators is unclear. Probably this was needed at some point during development of UP, but it seems unnecessary for the current version. The same can be said for the `NOT_IDENTICAL_Bool` operator.

immediately after the comment line. It is followed by a short motivation enclosed in curly brackets. In the original notation the motivations are informal or semi-formal English sentences; MathEdit, however, needs to store information about the derivation step in machine-readable format to be able to process derivations. We impose a simple syntax consisting of three main constructs on the bracketed text of a comment line: a *rule name*, a *rule application pattern* and a *substitution set*.

As an example consider the application of the following rule:

$$\text{DEMORGAN-1: } \neg P \vee \neg Q \equiv \neg(P \wedge Q)$$

in a simple derivation step, written in MathEdit notation[3]:

$$\begin{array}{l} A \wedge \underline{(\neg A \vee \neg B)} \\ \equiv \{\ \text{DEMORGAN-1 from left with } P := A,\ Q := B\ \} \\ A \wedge \neg(A \wedge B) \end{array}$$

Rule name (DEMORGAN-1): The editor identifies rules are by their names, so we store the name of the rule that was used as a simple text string within the comment line.

Application pattern (from left): The rule was applied from left-to-right. This means that the left-hand side of the rule $\neg P \vee \neg Q$ was unified with the indicated subexpression, and the result of the application is the right-hand side of the rule $\neg(P \wedge Q)$ with the substitution applied, $\neg(A \wedge B)$. In another step the rule could be used the other way around, with the right-hand side being unified with the subexpression and the result being the left-hand side. In both cases the relation symbol would be the equivalence sign. Left-to-right and right-to-left are the two most common rule application patterns, but the editor allows any number of patterns. The mathematical profile provides for a given rule a list of application patterns.

Substitution set (with $P := A,\ Q := B$): This is a comma-separated list of expressions, each describing a substitution pair. Special profile functions are provided to parse a substitution into a $v \mapsto e$ pair, where v and e are the variable and expression ASTs respectively. If there are ambiguous substitution pairs in the list, i.e., the same variable is substituted with two different expressions, the last pair in the list is used.

A reduction step can contain any number of subderivations. Subderivations in MathEdit look similar to those in structured calculational proofs and take

[3] In MathEdit derivations, comments do not line up with terms, but rather derivations have a flushed left margin. This is an implementation issue.

advantage of the outlining functionality of the editor. Subderivation are placed under the comment line of the step to which they belong and are indented one level deeper. This enables to user to hide all subderivations of a step by collapsing the comment line. MathEdit does not support labeling of subderivations, so the order of subderivations should match the order of rule hypotheses.

Potential assumptions are listed within angle brackets in the beginning of a subderivation and are formatted and parsed as rules. These are created when a conditional or focusing rule provides assumptions that can be used in a subderivation. Such rules are correct only in the context of the hypothesis required by the rule, and the scope of applicability is thus restricted to the subderivation and sub-subderivations down to any level.

A *simplification step* is a special kind of reduction step which can be used as an interface to external, "black-box" reduction tools. The comment line contains a set of rules, but without details about how to apply them. The external tool is sent a term and the list of rules, and returns a new term which is reduction of the original expression. The rules are applied according to the tool's own reduction strategies. Since such strategies can be time-consuming, a timeout parameter can be given in the comment line as a rough instrument to control the external tool.

6.2 Data Representation of Structured Derivations

The BNF grammar for structured derivations implemented in MathEdit is given in figure 2. Parsing is done in two sequential stages, *lexical* and *syntactic analysis*, by a routine based on the Python parser generator toolkit *Spark* [2]. Three different types of tokens are produced during the lexical analysis pass: identifiers (*IDENTIFIER*), integers (*INTEGER*), mathematical expressions (*MATHEXP*, *RULE*, *SUBEXP*) and special indentation "deepening" tokens (⌊⎵⌋). The tokenisation is based on string-matching regular expressions and markup information (for mathematical expressions). Deepening tokens are inserted into the token stream at points where the indentation level increases; this information is used in the parsing pass to identify where a subderivation starts.

Based on the token stream, the syntactic analysis pass generates a parse tree storing all elements of a derivation in a format optimized for programmatic access. Nodes in the tree have zero or more ordered children, and can be of eight different types:

DERIVATION Represents a derivation, either top-level or subderivation. The root node is of this type. Its children are, in order, an optional ASSUMPTIONLIST node, followed by at least one child of type TERM. After this comes zero or more repetitions of the following sequence:

⟨derivation⟩ ::= ⟨steplist⟩
| ⟨assumptionlist⟩ ⟨steplist⟩
| `"proof of"` *IDENTIFIER* ⌊⌋ ⟨steplist⟩

⟨derivationlist⟩ ::= ⟨derivation⟩
| ⟨derivation⟩ ⟨derivationlist⟩
| `"•"` ⟨derivationlist⟩

⟨steplist⟩ ::= ⟨term⟩
| ⟨term⟩ ⟨comment⟩ ⟨steplist⟩
| ⟨term⟩ ⟨comment⟩ ⟨term⟩
| ⟨term⟩ ⟨comment⟩ ⌊⌋ ⟨derivationlist⟩ ⟨steplist⟩

⟨assumption⟩ ::= `"⟨"` *RULE* `"⟩"`

⟨assumptionlist⟩ ::= ⟨assumption⟩
| ⟨assumption⟩ ⟨assumptionlist⟩
| `"•"` ⟨assumptionlist⟩

⟨comment⟩ ::= ⟨application⟩ | ⟨simplification⟩

⟨application⟩ ::= *IDENTIFIER* { *IDENTIFIER* }
| *IDENTIFIER* { *IDENTIFIER* ⟨patterndecl⟩ }
| *IDENTIFIER* { *IDENTIFIER* `"with"` ⟨substlist⟩ }
| *IDENTIFIER* { *IDENTIFIER* ⟨patterndecl⟩ `"with"` ⟨substlist⟩ }

⟨simplification⟩ ::= *IDENTIFIER* { `"simplification using"` ⟨rulelist⟩ `"maxsteps"` *INTEGER* }

⟨patterndecl⟩ ::= `"from left"` | `"from right"` | `"pattern"` *INTEGER*

⟨substlist⟩ ::= *MATHEXP* | *MATHEXP* `","` ⟨substlist⟩

⟨term⟩ ::= ⟨termexp⟩ | ⟨termexp⟩ ⟨termsubexp⟩

⟨termexp⟩ ::= *MATHEXP*

⟨termsubexp⟩ ::= *SUBEXP*

⟨rulelist⟩ ::= *IDENTIFIER* | *IDENTIFIER* `","` ⟨rulelist⟩

Figure 2: BNF grammar for structured derivations in MathEdit.

an APPLICATION or SIMPLIFICATION node followed by zero or more DERIVATION nodes, and a TERM node. This represents a proof (with possible subderivations), created using either a regular rule (APPLICATION), or the profile's `simplify` function (SIMPLIFICATION). Each subderivation is represented by a DERIVATION subtree.

APPLICATION Represents the "comment" in a derivation step in which a rule has been applied to a subexpression of a term to generate a new step. Nodes of this type store the name of the used rule and the application pattern (as an integer index into the list returned by the profile's **get rule patterns** method. Its only child is a single SUBSTLIST node.

SIMPLIFICATION A derivation step in which the profile's `simplify` has been used. Stores the relation symbol and the n argument sent to the function. Its only child is a single RULELIST node.

TERM A term in the derivation. Nodes of this type store two ASTs, for both the expression and subexpression on which a rule has been applied. It does not have any child nodes.

SUBSTLIST Stores the substitution set as a list of substitution ASTs.

RULELIST Stores a list of rule names used in a simplification step.

ASSUMPTION Represents an assumption (local rule), and stores its rule AST.

ASSUMPTIONLIST An ordered set of ASSUMPTION nodes.

6.3 Rule Application

The editor provides a feature that allows the user to select a subexpression and click a button to get an automatically generated menu of applicable rules. The user can then select a desired rule application and apply it, thereby generating a new derivation step. Rules can be applied in both forward (starting from the known) and backward (starting from the goal) directions. If the rule in use requires subderivations, the editor sets up an outline for each subderivation.

Displaying applicable rules requires gathering all defined rules and a means of testing if a rule is applicable. A rule is called *available* at a specific derivation step in a document if the editor knows about the rule (i.e., it has parsed the definition) at that point, and it is called *applicable* if it is both available and unifiable with the selected subexpression. Because rules are parsed into expression AST:s by the mathematical profile, a special method **get rule patterns** in the MPI is used to convert a rule AST into a list

of *patterns.* A pattern represents an explicit rule application and consist of a source expression, a relation symbol, a target expression and a list of hypotheses.

An important question is whether a rule should be available for use in derivations in the whole document or only in derivations after the line on which it was declared. It could be argued that since rules must be declared before they are used the scope of availability for a rule should be from the line of definition to the end of the document. A one-pass parsing would then be possible, and reading a document from top to bottom would ensure that no unknown rule is encountered. Nevertheless, in mathematical papers it is common to list rules and lemmas at the end or in a separate appendix. It was thus decided to not generally restrict the scope of rules, so a rule is always available everywhere in the document in which it was declared. However, one exception to the principle of universal rule scope are the special *local rules* based on assumptions in subderivations, as the availability of these rules is restricted to the subderivation in which they are declared.

MathEdit uses *unification* of the rule source expression with the expression being transformed to determine applicability. Unification identifies two symbolic expressions by binding the contents of variables to subexpressions. As an example, the expressions $s = x + y$ and $t = a + b \times c$ become identical if the substitution

$$\sigma = \{x \mapsto a, y \mapsto b \times c\}$$

is applied to s, i.e., x is replaced by a and y is replaced by $b \times c$. The substitution set σ is a *unifier* of the expressions. Application of a unifier to a term is written using postfix notation, i.e. $s\sigma = t$.

The MathEdit profile typically determines the *most general first-order* unifier. A unification function in the profile is called with the source side expression of a rule application pattern and the expression to be derived as parameters, and produces a result of either *nil*, meaning that unification was not possible, or a (possibly empty) substitution set. In the case of a non-*nil* result the rule pattern is deemed applicable, and the rule itself along with the target side (with substitutions applied) and the individual substitutions are displayed in the list of applicable rules. If several patterns of the same rule are applicable (rules may be applied in more than one way, typically left-to-right or right-to-left), all possible applications are displayed by the editor.

6.4 Derivation Checking

By *derivation checking* we mean the procedure in which the program checks the *well-formedness* of a derivation. A derivation is well-formed if it adheres

to the syntactic and semantic requirements on derivations and all rule applications are *valid* with the stated substitutions. This checking algorithm executes completely outside the mathematical profile and uses only the parsing, unparsing and unification interface of the profile, and thus has no access to intrinsic information about the mathematics in use. Derivation checking is not a formal verification of the correctness of the proof. The derivation checker can be likened with a compiler, which checks the syntax and semantics of source code but does not verify that the compiled program implements its specifications.

Implementing a proof system is not the goal of MathEdit. Existing dedicated *theorem proving assistants* with a long-standing reputation of reliability, such as PVS and the HOL system, are the most suitable tools for formal proving. Integration with such tools would be useful to provide an independent assertion of the correctness of MathEdit derivations, but that is outside the scope of this paper. Experiments have been done with HOL integration, but no complete implementation exists as of yet.

The derivation checking algorithm as implemented in MathEdit is shown in Algorithm 1. It processes a sequence of derivations; a sequence is either the set of top-level derivations in the document being checked, or the subderivations of some step. The latter case occurs when the algorithm finds a step with subderivations and is applied recursively. If any one of the assertions fail, the algorithm terminates.

In MathEdit it is also possible to associate a proof with a specific rule. A trivial kind of checking is performed to detect cycles in such declarations; existence of a cycle is an error, since a proof of a rule must not rely on the correctness of the rule being proved. However, it should be noted that MathEdit does not implement a full proof verification system. The relation between terms is only checked on a per-step basis, and there is no attempt to verify that the composition of relations matches the relation of the proved rule or hypothesis.

7 Working with MathEdit

In this section we first give a general outline the MathEdit workflow, and follow up with examples which show how the powerful extensible syntax of the UP and Math Lib profiles makes it possible to define new mathematical languages on the fly.

7.1 Workflow

Producing a MathEdit document is an iterative process in which the user constantly works within the same framework in a modification-feedback loop.

Algorithm 1 Derivation checking

Loop for each derivation *der* in the sequence:

Parse *der*. This also requires parsing all subderivations and mathematical expressions in *der*. If a syntax error occurs, report it and terminate

Loop for each step *step* in *der*:

Let t_{from} and t_{to} be the terms before and after the transformation respectively. Let s_{from} be the denoted subexpression of t_{from}

If *step* is of type APPLICATION:

step then contains the 4-tuple $\langle \alpha, r, i, \sigma \rangle$ where α is the relation symbol, r is the rule name, i is the rule application pattern index and σ is the substitution set

Check that a rule named r exists

Get the i:th rule application pattern p of the rule r on s_{from}. $p = \langle H, r_{from}, \alpha', r_{to} \rangle$. $H = [\langle h_1, A_1 \rangle, \langle h_2, A_2 \rangle, ...]$ is a list of hypotheses. Let A_k be the set of assumptions available for proving h_k

For each hypothesis h_k check that the matching subderivation d_k uses no other assumptions than those declared in A_k

Apply this algorithm recursively to the list of subderivations

Check that $\alpha = \alpha'$

Unify r_{from} with s_{from} and let σ' be the resulting substitution set. Let ω be the set of free variables, i.e. unbound variables in r_{to}. Check that $\{x \mapsto e | x \mapsto e \in \sigma \wedge x \notin \omega\} = \sigma'$

Check that $t_{to} = t_{from}[s_{from} \mapsto r_{to}\sigma]$, i.e. the resulting expression should be equal to the initial expression where the subexpression s_{from} has been replaced with the result of applying the rule r to s_{from}

Else if *step* is of type SIMPLIFICATION:

step then contains $\langle \alpha, m, R \rangle$ where α is a relation symbol, m is the maximum number of steps allowed and $R = [r_1, ..., r_n]$ is the sequence of rules used in the simplification

Check that rules $r_1, ..., r_n$ exist

Check that a simplification can be obtained by calling the profile's simplification function on s_{from} with arguments m and R. Let the result be $\langle \alpha', s' \rangle$

Check that $\alpha = \alpha'$

Check that $t_{to} = t_{from}[s_{from} \mapsto s']$

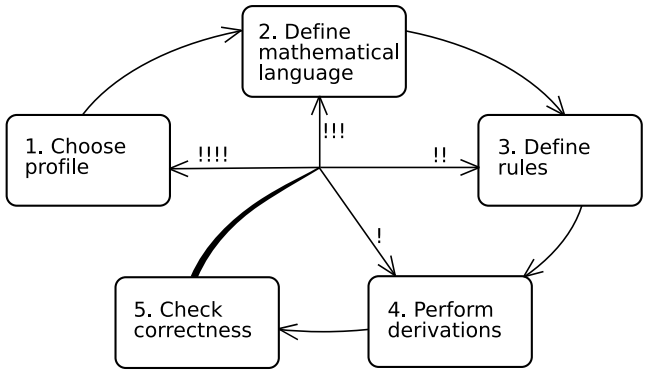

Figure 3: Basic MathEdit workflow

Figure 3 describes the basic stages of this process. The focus is on the mathematical features of the editor and not on general text editing, so stages like formatting, saving, etc. are not considered.

The user starts by creating a new document and choosing a mathematical profile (1). Currently the choice is between UP and Math Lib, and considering that Math Lib is an extension of UP with additional features, users are recommended to use Math Lib for most purposes. However, a user who wants to define a mathematical language from the ground up might be better off with the bare-bones UP to avoid clashes with existing definitions in Math Lib.

The user may want to define a mathematical language (2) and/or a set of rules (3). The amount of work done in these steps varies depending on the user's objective: and advanced user building up a theory from the ground or creating a prelude for others to use might spend a lot of time in these stages, while a user focusing on doing derivations within a ready-made environment, such as Math Lib, might not.

When there is at least one rule available (built-in or user-defined), derivations can be performed (4). The user enters a formula and marks it as a mathematical expression, after which it becomes subject to parsing and type-checking. If the expression is valid, the user can select any subexpression and ask the editor to show a list of applicable rules (menu: Derivation ▷ Derive). Applying a rule starts a new derivation and adds a first step. Steps can also be added manually, although this requires some familiarity with the derivation syntax in figure 2 from the user.

After a number of steps, the user might conclude that the derivation is finished and proceed to the correctness checking stage (5). Either all derivations in the document (menu: Derivation ▷ Check All Derivations) or a specific derivation (menu: Derivation ▷ Check Current Derivation) can be checked; the program then applies the algorithm in figure 1. Erroneous lines in a checked derivation are marked and errors details are reported in a

window, allowing the user to view the list of errors and go to any one of the offending lines.

When performing a derivation the need for a new rule may arise. The user might also want to change an existing rule, e.g., generalizing it. Similarly, the mathematical language defined might not be expressive enough to proceed and needs to be changed, e.g., an operator could be missing. However, changes in rules might invalidate derivations, and changes in mathematical language might invalidate expressions, rules and derivations. The earlier the level at which the change occurs, the more consequences it will have: this is illustrated in the figure by an increasing amount exclamation marks on the paths of the last transition. At the extreme, switching to another mathematical profile with a different syntax invalidates every definition, expression, rule and derivation in the document.

7.2 Using UP

In this section we explain how UP can be used to create you own mathematical language. We will do this by defining a theory for parallel processes in the style of ACP [14, 12, 19]. In addition to introducing types and operators for parallel processes, we define the axioms for the process theory and show how these axioms can be used as rules to create structured derivations.

7.2.1 ACP

We will distinguish three types: *Action*, *Atom*, and *Process*. The actions form the basic building blocks and are usually defined with a particular application in mind. The atoms include the actions and, in addition, a predefined deadlock constant δsymbolizing inaction (the absence of an action). The processes include all atoms and, in addition, the compound processes build up from atoms and process operators. The three types are defined as follows.

```
op ActionAtom: Action → Atom .
op AtomProcess: Atom → Process .
op ProcessTerm: Process → Term .
```

These are three invisible (no syntax) operators that define the following subtype relations.

$$\begin{aligned} \mathit{Action} &\subseteq \mathit{Atom} \\ \mathit{Atom} &\subseteq \mathit{Process} \\ \mathit{Process} &\subseteq \mathit{Term} \end{aligned}$$

As mentioned above, actions are usually defined with a particular application in mind. Therefore, we postpone the definition of concrete actions for now. The deadlock constant is defined:

```
op Deadlock: "δ" →Atom .
```

The theory has two operators for sequential processes and three for parallel processes. The sequential operator are defined as follows.

```
op Alt: (Process) "\+" Process → Process [prec=100] .
op Seq: (Process) "·" Process → Process [prec=101] .
```

The `Alt` operator puts two processes in alternative composition. The syntax of the operator is defined by the regular expression `"\+"` which matched the single character `+`. The arguments of this operator should be of type `Process`. To reduce the number of required parentheses in processes, we have enclosed the first argument type in parentheses. This means that UP will add parentheses to the left argument whenever needed to get a correct parse. Consequently, when we write $\delta+\delta+\delta$, UP will read this as $(\delta+\delta)+\delta$. The `prec`-attribute is set to 100.

The `Seq` operator's definition is similar to that of `Alt`. It's syntax is a $\cdot$ (centered dot), and since its `prec`-attribute is 101, it binds stronger than the `Alt` operator. Therefore, an expression $\delta \cdot \delta + \delta$ is parsed as $(\delta \cdot \delta) + \delta$.

Before we continue with the definitions of operators to construct parallel processes, we first give the axioms for the sequential process operators introduced so far. To define these rules, we need three variables of type *Process*:

```
var x,y,z : Process .
```

Now, x, y, and z stand for arbitrary processes. In addition to using identifiers to define variables, it is possible to use double-quoted regular expressions. For instance, the following line defines an infinite set of variable consisting of one lower case letter followed by zero or more primes:

```
var "[a-z]'*" : Process .
```

The axioms of the sequential process operators are defined in the following rules.

```
rule Alt_Delta: x+δ= x .
rule Delta_Seq: δ·x = δ .
rule Comm_Alt: x+y = y+x .
rule Assoc_Alt: (x+y)+z = x+(y+z) .
rule Assoc_Seq: (x·y)·z = x·(y·z) .
rule Alt_Seq_RDistr: x·z + y·z = (x+y)·z .
```

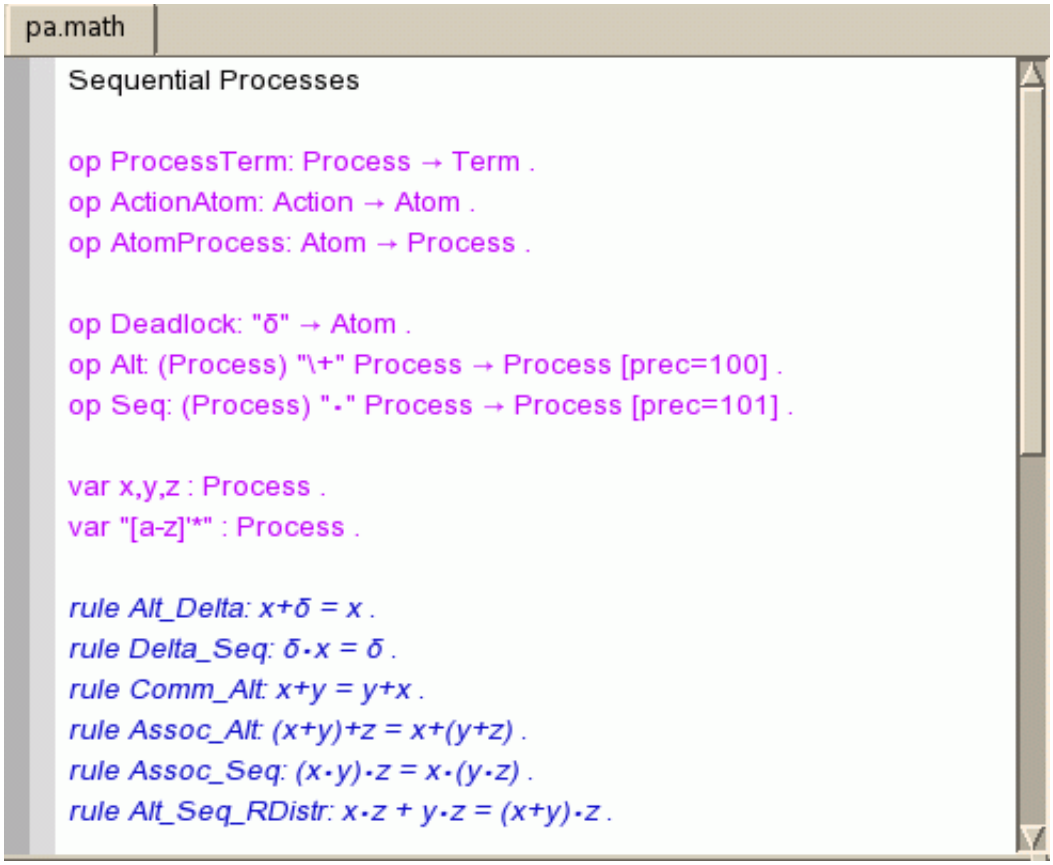

Figure 4: Operators and rules for sequential processes.

The `Alt_Delta` rule says that a choice, indicated by the + operator, between deadlock and another process is not really having a choice, because the deadlock will never be chosen. In other words, δ is a neutral-element for the + operator. The next rule, `Delta_Seq`, says that nothing follows deadlock, that is, δ is a left-zero-element for the $\cdot$ operator . `Comm_Alt` and `Assoc_Alt` define commutativity and associativity of the + operator and `Assoc_Seq` defines associativity of the $\cdot$ operator. Finally, `Alt_Seq_RDistr` defines the (right) distributivity of $\cdot$ over +. Figure 4 shows the text canvas of MathEdit with the operators and rules described so far.

Next, we introduce the parallel composition operator, which is called `Merge`.

```
op Merge: (Process) "||" Process →Process [prec=92] .
```

The `Merge`-operator runs two processes in parallel allowing them to operate autonomously or interactively. This means that the actions of the processes are interleaved (merged) arbitrarily or they are synchronized into communication actions. The merge operator is defined by the following rule.

```
rule Merge_Def: x‖y = (x⊩y) + (y⊩x) + (x|y) .
```

Intuitively, the parallel composition of `x` and `y` (denoted by `x‖ y`) can perform an action from `x` (denoted by `x⊩y`), an action from `y` (denoted by `y⊩x`), or a communication between `x` and `y` (denoted by `x|y`). The `⊩` and the `|` operators will be defined shortly. In addition to this rule, we define two rules that express the commutativity and associativity of the `Merge`-operator.

```
rule Comm_Merge: x‖y = y‖x .
rule Assoc_Merge: (x‖y)‖y = x‖(y‖y) .
```

The definitions of the `CommMerge` and `LeftMerge`-operators are as follows.

```
op CommMerge: (Process) "\|" Process → Process [prec=90].
op LeftMerge: (Process) "⊩" Process →Process [prec=92].
```

In ACP, communication is defined on the level of actions and communication between processes is defined in terms of the actions these processes are built up from. Therefore, to give the rules for the `CommMerge` operator, we need variables ranging over actions. However, it turns out that the special atom δ behaves almost as an action with respect to the `CommMerge` operator. To keep the number of rules small, we therefore define variables of type `Atom`.

```
var "[abc]'*" →Atom .
```

This defines infinitely many variables of type `Atom` ($=$ `Action` $\cup \{\delta\}$) . The variables start with an `a`, `b`, or `c`, and end with zero or more `'` (prime) symbols. The rules for the `CommMerge` are defines as follows.

```
rule CommMerge_Delta: x|δ = δ.
rule Delta_CommMerge: δ|x = δ.
rule AtomPrefix_CommMerge_AtomPrefix: a·x | b·y = (a|b)·(x‖y) .
rule AtomPrefix_CommMerge_Atom: a·x | b = (a|b)·x .
rule Atom_CommMerge_AtomPrefix: a | b·y = (a|b)·y .
rule Comm_CommMerge: x|y = y|x .
rule Assoc_CommMerge: (x|y)|z = x|(y|z) .
rule Alt_CommMerge: (x+y)|z = (x|y) + (y|z) .
```

The `LeftMerge`-operator is an auxiliary operator needed to give a finite axiomatisation of the `Merge`-operator. It behaves essentially equal to the `Merge`-operator, except that its first action has to come from its left argument. The rules for the `LeftMerge`-operator are as follows.

```
rule Atom_LeftMerge: a⌊⊢x = a·x .
rule AtomPrefix_LeftMerge: a·x⌊⊢y = a·(x‖y) .
rule Alt_LeftMerge: (x+y)⌊⊢z = (x⌊⊢z) + (y⌊⊢z) .
```

A powerful proof technique of ACP is *basic term induction*. *Basic terms* are defined inductively as follows.

1. δ is a basic term;
2. all $a \in$ `Action` are basic terms;
3. if s is a basic term and $a \in$ `Action`, then $a \cdot s$ is a basic term;
4. if s and t are basic terms and neither of them is δ, then $s + t$ is a basic term.

It can be proved that every `Process` built up from the process operators introduced so far is equal to a basic term. Therefore, by proving properties about basic terms, we can establish properties about all processes. As basic terms are defined inductively, we can use a structural induction technique to prove properties about basic terms. This technique is called *basic term induction*:[4]

Basic term induction *Let X and Y be two processes and let x be a sub-process of X and possibly of Y. If the following properties hold, then $X = Y$.*

1. $X[\delta/x] = Y[\delta/x]$;
2. $X[a/x] = Y[a/x]$ *for a an* `Action`;
3. $X[a \cdot s/x] = Y[a \cdot s/x]$ *for a an* `Action` *and s a basic term such that* $X[s/x] = Y[s/x]$;
4. $X[(s+t)/x] = Y[(s+t)/x]$ *for s and t a basic terms such that* $X[s/x] = Y[s/x]$ *and* $X[t/x] = Y[t/x]$.

If we apply this technique to prove $X = Y$, we say we prove $X = Y$ by (basic term) induction on x. Note that it is not strictly needed for x to be a sub-process of X (or Y); if it is not, proving $X = Y$ by basic term induction on x boils down to proving $X = Y$ directly.

It is possible to define (structural) induction in UP, although we do note that the current implementation of UP does not provide all features we would

[4] In general, basic term induction can be used to prove any property $P(X_1, \ldots, X_n)$ of processes $X_1, \ldots, X_n$. Here, the property is the binary relation $P(X, Y) \hat{=} (X = Y)$.

like it to have as far as induction is concerned. Anyway, the current implementation comes a long way and it is worthwhile to illustrate this. The following rule is a rather direct translation of the basic term induction definition given above.

```
rule BTI: [X[x :=  δ]=Y[x :=  δ],
           X[x :=  a] = Y[x :=  a],
           [ X[x :=  s] = Y[x :=  s ]
             ⊢ X[x := a·s] = Y[x :=  a·s],
           [ X[x :=  s] = Y[x :=  s],
             X[x :=  t] = Y[x :=  t] ]
             ⊢ X[x :=  s+t] = Y[x :=  s+t],
          ]⊢X = Y .
```

Although this rule looks fine, UP cannot guarantee that we will use it correctly. The problem is that when we instantiate this rule, we are not allowed to assume anything about `a`, `s`, and `t` (except what is given by the basic term induction rule). For instance, if we apply this rule in a derivation in which any of these three symbols already occurs, we run the risk of using properties about those occurrences and UP will not warn us about it. Such situations can easily arise. For instance, when we try to prove $X = Y$ we might first do basic term induction on a x and then a *nested* basic term induction on a y. In the nested step, we have to make sure `a` is not equal to the `a` of the outer step.

We will now use UP to prove that $x \parallel \delta = x \cdot \delta$ by basic term induction on x. This prove has been done completely in MathEdit; it is a derivation of 100 lines. Consequently, MathEdit is able to check the validity of the proof. Below, we have split up the proof in several parts and comment each part separately. The first part is the whole proof without the subproofs resulting from application of the `BTI` rule. The start and end of the subproofs are given, but the details are left out, exactly as MathEdit does when you apply a rule with proof obligation (side conditions).

```
  x‖δ
= {Merge_Def from left with x:=x, y:=δ}
  (x|⊢δ)+(δ|⊢x)+(x|δ)
= {BTI from left with X:=x|⊢δ, Y:=x·δ, a:=a, s:=s, t:=t, x:=x}
  • (x|⊢δ)[x:=δ]
  = { ... }
    (x·δ)[x:=δ]
  • (x|⊢δ)[x:=a]
  = { ... }
    (x·δ)[x:=a]
  • ⟨assumption2_2_1:(x|⊢δ)[x:=s] = (x·δ)[x:=s]⟩
```

```
    (x|⊢δ)[x:=a·s]
  = { ... }
    (x·δ)[x:=a·s]
  • ⟨assumption2_3_1:(x|⊢δ)[x:=s] = (x·δ)[x:=s]⟩
    ⟨assumption2_3_2:(x|⊢δ)[x:=t] = (x·δ)[x:=t]⟩
    (x|⊢δ)[x:=s+t]
  = { ... }
    (x·δ)[x:=s+t]
··· x·δ+(δ|⊢x)+(x|δ)
= { Atom_LeftMerge from left with a:=δ, x:=x }
  x·δ + δ·x + (x|δ)
= { CommMerge_Delta from left with x:=x }
  x·δ + δ·x + δ
= { Alt_Delta from left with x:=x·δ + δ·x }
  x·δ + δ·x
= { Delta_Seq from left with x:=x }
  x·δ + δ
= { Alt_Delta from left with x:=x·δ }
  x·δ
```

We see from this part that MathEdit creates the expected subproofs: first we have to prove the property for `x=`δ, then for `x=a`, then for `x=a·s`, and finally for `x=s+t`. Furthermore, we get some assumptions for `s` and `t`. These assumptions can be used as normal rules in their corresponding subproofs, as will be shown later. The steps after the BTI step are rather trivial. In fact, these steps can be done as one simplification step.

7.2.2 ACP example

As an example, we will define the behavior of a web server as an expression of the type *Process*. The actions for the web server are *RecvReq*, *GetPage*, *SendPage*, and *SendErrorPage*. A rather abstract view of a web server can be defined as follows.

$$\begin{aligned} Server &= RecvReq \cdot (Server \parallel HandleRequest) \\ HandleRequest &= GetPage \cdot (SendPage + SendErrorPage) \end{aligned}$$

The *Server* waits for a request for a certain web page (*RecvReq*). When it has received a request, it starts up a new *Server*, to deal with following requests, and in parallel to that it starts handling the request (*HandleRequest*). Handling of a request means retrieving the requested web page (*GetPage*) and, when it exists, sending it back *(SendPage*), or, when it does not exist, sending an error page back (*SendErrorPage*).

To define the *Server* process in MathEdit, we first define the actions.

Figure 5: A simple web server.

```
op "RecvReq" → Action .
op "GetPage" →Action .
op "SendPage" →Action .
op "SendErrorPage" → Action .
```

Note that these operator definitions are without a name. As the syntax of the operators are perfectly good names, it seems unnecessary to add names explicitly. Behind the scenes, UP will create internal names for these operators. Such internal names have the form "`op_`N", where N is a sequence number.

Next, we define the compound processes *Server* and *HandleRequest*. Both consist of an operator definition and a defining rule.

```
op "Server" → Process .
op "HandleRequest"→ Process .
rule ServerDef:
    Server = RecvRequest·(Server ‖ HandleRequest) .
rule HandleRequestDef:
    HandleRequest = GetPage·(SendPage + SendErrorPage) .
```

Note that rules always need a name, even if they are as simple as the two rules defining `Server` and `HandleRequest`. The reason for this is that MathEdit needs to refer to these rules while creating or checking derivations. Figure 5 shows the operators and rules of the web server as they appear on the MathEdit text canvas.

7.3 Using Math Lib

Math Lib has built-in support for expressions including Boolean connectives, quantifiers, and arithmetic expressions. However, no constructs for reasoning explicitly about sets is provided, so we describe how this addition can be made to the Math Lib profile with appropriate declarations contained in the same document as the proof itself. The following example builds a simple mathematical language to describe sets, and introduces definitions of set comprehension and the operators union and intersection based on set membership. Set comprehension allows us to reason about a Boolean variable quantified over the elements of a set; since membership is Boolean-valued, we can use the existing Math Lib support for Booleans.

We start by introducing two new types, *Elem* and *Set*, which are both subtypes of the built-in type *Term*. This means that they can be used as top-level terms in expressions. We might also wish to make *Set* a subtype of *Elem*, so that it is possible to have sets of sets. This is achieved by entering the following lines in an empty document using the Math Lib profile and formatting each line as a *definition:*

```
op ElemTerm: Elem→Term .
op SetTerm: Set→Term .
op SetElem: Set→Elem .
```

The *op* keyword is also used to define operators. We now define the operators for set membership, union, intersection and comprehension:

```
op IN: Elem "∈" Set→Bool [prec=50] .
op UNION: Set "∪" Set→Set [prec=100,commutative] .
op INTERSECTON: Set "∧" Set→Set [prec=100,commutative] .
op SETCOMP: "{" Elem "\|" Bool "}" → Set .
```

The strings within quotation marks are regular expressions used by Math Lib's scanner/parser to recognize operations. Binary operators can be associated with precedence and commutativity properties, which enable us to omit superfluous parentheses in expressions. Also note the use of the predefined *Bool* type.

To be able to define rules and expressions with variables, we define a number of unprimed and primed set, element and Boolean variables:

```
var "S[']*": Set .
var A,B,C: Set .
var e,v: Elem .
var "b[']*": Bool .
```

We now define rules for rewriting union/intersection using Boolean disjunction/conjunction and for introducing set comprehension. Rules are declared using a different formatting than definitions, and the text is colored blue in the editor:

```
rule UnionDef: e∈S∪S' ≡ (e∈S)∨(e∈S') .
rule IntersectionDef: e∈S∩S' ≡ (e∈S)∧(e∈S') .
rule SetComprehension: S = {e|e∈S} .
```

Math Lib does not provide a built-in rule for distributing disjunction over conjunction, so we need to define this. To be able to transform only the predicate of a set comprehension construct in a subderivation, we also define a focusing rule:

```
rule DistrDisjOverConj: b∨(b'∧b'') ≡ (b∨b')∧(b∨b'') .
rule FocusOnPredicate: {v|b} = {v|b'} if b≡b' .
```

The precondition clause $b \equiv b'$ in the second rule triggers the creation of a subderivation when the rule is applied. When applied from left to right, b' is a free variable which can be assigned any expression.

With these definitions, it is now possible to perform the following derivation:

$\underline{A \cup (B \cap C)}$
$=$ { SetComprehension from left with $S := A \cup (B \cap C)$, $e := v$ }
$\underline{\{v | v \in A \cup (B \cap C)\}}$
$=$ { FocusOnPredicate from left with $b' := v \in (A \cup B) \cap (A \cup C)$, $b := v \in A \cup (B \cap C)$, $v := v$ }

- $\underline{v \in A \cup (B \cap C)}$
  $\equiv$ { UnionDef from left with $S' := B \cap C$, $S := A$, $e := v$ }
  $(v \in A) \vee \underline{(v \in B \cap C)}$
  $\equiv$ { IntersectionDef from left with $S' := C$, $S := B$, $e := v$ }
  $\underline{(v \in A) \vee (v \in B) \wedge (v \in C)}$
  $\equiv$ { DistrDisjOverConj from left with $b'' := v \in C$, $b' := v \in B$, $b := v \in A$ }
  $\underline{((v \in A) \vee (v \in B))} \wedge ((v \in A) \vee (v \in C))$
  $\equiv$ { UnionDef from right with $S' := B$, $S := A$, $e := v$ }
  $(v \in A \cup B) \wedge \underline{((v \in A) \vee (v \in C))}$
  $\equiv$ { UnionDef from right with $S' := C$, $S := A$, $e := v$ }
  $\underline{(v \in A \cup B) \wedge (v \in A \cup C)}$
  $\equiv$ { IntersectionDef from right with $S' := A \cup C$, $S := A \cup B$, $e := v$ }
  $v \in (A \cup B) \cap (A \cup C)$

$\cdots$ $\underline{\{v|v \in (A \cup B) \cap (A \cup C)\}}$

= { SetComprehension from right with $S := (A \cup B) \cap (A \cup C)$, $e := v$ }

$(A \cup B) \cap (A \cup C)$

Each step in the derivation can be typed in manually or created by using MathEdit's "Show applicable rules" button to list the applicable rules in each step and selecting the appropriate rule. A screenshot showing the widget listing applicable rules for the *FocusOnPredicate* step can be seen in figure 6. Top-level nodes in the tree represent rules while the children are possible applications. An application can be further expanded to reveal substitutions. The question marks in the substitution of variable b' indicate that in order to apply the focusing rule the user must supply a value for the free variable b', which in this case is $v \in (A \cup B) \cap (A \cup C)$.

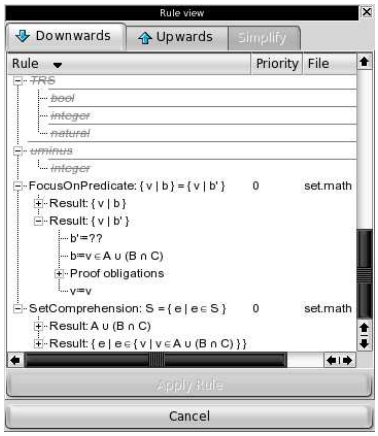

Figure 6: MathEdit's applicable rules list

8 Conclusion and Future Work

This paper has presented MathEdit, a tool for writing mathematical derivations in the structured calculational proof format. Since this format is distinguished from ordinary calculational proofs by its ability to hierarchically decompose proofs into smaller ones, it makes an outlining editor a natural choice of editing environment. By defining a syntax for derivations the editor provides a framework for representing and manipulating derivations, and by providing an extensible profile interface, it enables users to reason about many different kinds of mathematics. Although the functionality offered by MathEdit is limited compared to that of more specialized programs,

the freedom and extensibility offered is valuable as a basis for future development and research; since there are no built-in assumptions about a specific mathematical theory, new ideas can be tested in the editor rather freely.

The experience gathered from developing and using MathEdit has resulted in a number of important insights. There are still some improvements needed to make it useful for people working with mathematics in practice. One of high importance is notation; currently MathEdit is completely line-based and does not allow nice-looking multi-tiered typesetting for constructs such as rationals, square roots and matrices. It would certainly be beneficial for the usability and aesthetics of MathEdit if such constructs could be displayed and edited, and preferably encoded in a standard format such as MathML [26]. However, specifying and implementing an intuitive user interface for more advanced mathematical constructs is no small feat. An alternative solution, which we are currently pursuing, is to take an existing mathematics editor with an extensible document structure, such as TEXmacs, and implement an integration layer to either our own parser/proof engine or to a more powerful theorem prover.

Currently the user is required to invent the proofs for theorems; the editor only assists in the task of writing them. A future version of MathEdit could make use of an advanced theorem proving assistant to create proofs as well as perform the correctness checking. Progress has been made in this area, as integration layers for HOL [18] and *Simplify* [10] have been partially implemented.

Proof browsing in hypertext media is an interesting technique for proof presentation that is worth developing beyond the limited export filters that MathEdit currently implements. The user should be able to publish a *work book*, consisting of a number of MathEdit documents, in both hypertext and printed formats. A template system could be devised to ensure that workbooks can conform to different styles.

References

[1] *The Unicode Standard, Version 4.0.* Addison-Wesley Longman Publishing Co., Inc., 2003.

[2] J. Aycock. Compiling little languages in Python, 1998.

[3] Ralph Back, Jim Grundy, and Joakim von Wright. Structured calculational proof. *Formal Aspects of Computing*, 9(5–6):469–483, 1997.

[4] Ralph-Johan Back. Software construction by stepwise feature introduction. In *ZB '02: Proceedings of the 2nd International Conference of B and Z Users on Formal Specification and Development in Z and B*, pages 162–183. Springer-Verlag, 2002.

[5] Ralph-Johan Back, Johannes Eriksson, and Luka Milovanov. Experience on using stepwise feature introduction in software construction. Technical Report 705, TUCS, Turku, Finland, Aug 2005.

[6] Ralph-Johan Back and Pentti Hietala. I3v: A program proof management system. Report A136, University of Tampere, Department of Mathematical Sciences, 1986.

[7] Ralph-Johan Back, Piia Hirkman, and Luka Milovanov. Evaluating the XP customer model and design by contract. In Ralf Steinmetz and Andreas Mauthe, editors, *Proceedings of the 30th EUROMICRO Conference*, pages 318–325, Rennes, France, Aug 2004. IEEE.

[8] Ralph-Johan Back, Luka Milovanov, and Ivan Porres. Software development and experimentation in an academic environment: The Gaudi experience. Technical Report 641, TUCS - Turku Centre for Computer Science, Turku, Finland, Nov 2004.

[9] Ralph-Johan Back, Luka Milovanov, Ivan Porres, and Viorel Preoteasa. An experiment on extreme programming and stepwise feature introduction. Technical Report 451, TUCS - Turku Centre for Computer Science, Turku, Finland, Dec 2002.

[10] Ralph-Johan Back and Magnus Myreen. Tool support for invariant based programming. Technical Report 666, TUCS - Turku Centre for Computer Science, Turku, Finland, Feb 2005.

[11] Roland Carl Backhouse, Richard Verhoeven, and Olaf Weber. Math/pad: A system for on-line preparation of mathematical documents. *Software - Concepts and Tools*, 18(2):80–, 1997.

[12] J.C.M. Baeten and W.P Weijland. *Process Algebra*. Number 18 in Cambridge Tracts in Theoretical Computer Science. Cambridge University Press, 1990.

[13] K. Beck. *Extreme Programming Explained: Embrace Change*. The XP Series. Addison-Wesley, 1999. BEC k2 01:1 1.Ex.

[14] J.A. Bergstra and J.W. Klop. Process algebra for synchronous communication. *Information and Control*, 60(1/3):109–137, 1984.

[15] Michael Butler, Jim Grundy, Thomas Långbacka, Rimvydas Ruksenas, and Joakim von Wright. The refinement calculator: Proof support for program refinement. In Lindsay Groves and Steve Reeves, editors, *Formal Methods Pacific'97: Proceedings of FMP'97*, Discrete Mathematics & Theoretical Computer Science, pages 40–61, Wellington, New Zealand, July 1997. Springer-Verlag.

[16] B. W. Char, K. O. Geddes, G. H. Gonnet, B. L. Leong, M. B. Monagan, and S. M. Watt. *Maple V Language Reference Manual*. Springer-Verlag, 1991.

[17] Johannes Eriksson. Development of a mathematical derivation editor. Master's thesis, Åbo Akademi University, Department of Computer Science, 2004.

[18] Peter Eriksson. Integration of a mathematical derivation editor with HOL. Master's thesis, Åbo Akademi University, Department of Computer Science, 2004.

[19] Wan Fokkink. *Introduction to Process Algebra*. Springer, 2000.

[20] Gerhard Gentzen. *Untersuchungen über das logische Schließen*. Wissenschaftliche Buchgesellschaft Darmstadt, 1934.

[21] David Gries and Fred B. Schneider. *A logical approach to discrete math*. Springer-Verlag New York, Inc., New York, NY, USA, 1993.

[22] Jim Grundy. A browsable format for proof presentation. *Mathesis Universalis*, 1(2), Spring 1996.

[23] Cliff B. Jones and Richard Moore. Muffin: a user interface design experiment for a theorem proving assistant. In *Proceedings of the 2nd VDM-Europe Symposium on VDM—The Way Ahead*, pages 337–375. Springer-Verlag New York, Inc., 1988.

[24] Leslie Lamport. How to write a proof. *American Mathematical Monthly*, 102:600–608, 1995.

[25] Peter J. Robinson and John Staples. Formalizing a hierarchical structure of practical mathematical reasoning. *J. Log. Comput.*, 3(1):47–61, 1993.

[26] Pavi Sandhu. *The MathML Handbook*. Charles River Media, 2002.

[27] J. van der Hoeven. GNU TeXmacs: A free, structured, wysiwyg and technical text editor. In Daniel Flipo, editor, *Le document au XXI-ième siècle*, volume 39–40, pages 39–50, Metz, 14–17 mai 2001. Actes du congrès GUTenberg.

[28] Guido van Rossum and Fred L. Jr. Drake. *The Python Tutorial - An Introduction to Python*. Network Theory Ltd., 2003.

[29] Stephen Wolfram. *Mathematica: a system for doing mathematics by computer*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1988.

Turku
Centre *for*
Computer
Science

Lemminkäisenkatu 14 A, 20520 Turku, Finland | www.tucs.fi

University of Turku
- Department of Information Technology
- Department of Mathematics

Åbo Akademi University
- Department of Computer Science
- Institute for Advanced Management Systems Research

Turku School of Economics and Business Administration
- Institute of Information Systems Sciences

ISBN 978-952-12-2006-7
ISSN 1239-1891