Status Report on the Tight Integration of a Scientific Text-Editor and a Proof Assistance System

Serge Autexier

DFKI GmbH & Fachbereich Informatik, Universität des Saarlandes
66041 Saarbrücken, Germany (www.dfki.de/~serge)

Marc Wagner

Fachbereich Informatik, Universität des Saarlandes
66041 Saarbrücken, Germany (www.ags.uni-sb.de/~mwagner)

Abstract

In order to foster the use of proof assistance systems, we integrated the proof assistance system ΩMEGA with the standard scientific text-editor TEXMACS. We aim at a document-centric approach to formalizing and verifying mathematics and software. Assisted by the proof assistance system, the author writes her document entirely inside the text-editor in a language she is used to, that is a mixture of natural language and formulas in LATEX style. In this paper we briefly describe the PLATΩ-system that realizes the integration and the mechanism that allows the author to define her own notation inside a document in a natural way, which is used to parse the formulas written by the author as well as to render the formulas generated by the proof assistance system. We illustrate the different facets of the support offered to the author by the proof assistance system by giving a worked example.

1 Introduction

The vision of a powerful mathematical assistance environment that provides computer-based support for most tasks of a mathematician has stimulated new projects and international research networks in recent years across disciplinary boundaries. Even though the functionalities and strengths of proof assistance systems are generally not sufficiently developed to attract mathematicians on the edge of research, their capabilities are often sufficient for applications in e-learning and engineering contexts. However, a mathematical assistance system that shall be of effective support has to be highly user-oriented. We believe that such a system will only be widely accepted by users if the communication between human and machine satisfies their needs, in particular only if the extra time spent on the machine is by far compensated by the system support. One aspect of the user-friendliness is to integrate formal modeling and reasoning tools with software that users routinely employ for typical tasks.

One standard activity in mathematics and the areas that are based on mathematics is the preparation of documents using some standard text preparation system like LATEX.

This paper is electronically published in
Electronic Notes in Theoretical Computer Science
URL: www.elsevier.nl/locate/entcs

TEXMACS [20] is a scientific text-editor in the WYSIWYG paradigm that provides professional type-setting and supports authoring with powerful macro definition facilities like those in LATEX. As a first step towards assisting the authoring of mathematical documents, we integrated the proof assistance system ΩMEGA into TEXMACS using the generic mediator PLATΩ [22]. In this setting the formal content of a document must be amenable to machine processing, without imposing any restrictions on how the document is structured, on the language used in the document, or on the way the document can be changed. The PLATΩ system [21] transforms the representation of the formal content of a document into the representation used in a proof assistance system and maintains the consistency between both representations throughout the changes made on either side.

Such an integrated authoring environment should allow the user to write her mathematical documents in the language she is used to, that is a mixture of natural language and formulas in LATEX style with her own notation. To understand as far as possible the semantics of full natural language in a mathematical document we currently rely on annotations for the document structure that must be provided manually by the user. Although it might still be acceptable for an author to indicate the macro-structures like theories, definitions and theorems, writing annotated formulas (e.g. "`\F{in}{\V{x},\F{cup}{\V{A},\V{B}}}`" instead of "`x \in A \cup B`") is definitely not. We present a mechanism that allows authors to define their own notation and to use it when writing formulas within the same document. Furthermore, this mechanism enables the proof assistance system to access the formal content and use the same notation when presenting formulas to the author.

The paper is organized as follows: Section 2 gives a short introduction to the proof assistance system ΩMEGA, its main components MAYA and the TASK LAYER, as well as to the PLATΩ system that realizes the integration with the text-editor TEXMACS. Section 3 presents the annotation language for documents of the PLATΩ system, in particular for formulas. Inspired by notational definitions in text-books, we present the means the author has to define notations, from which an *abstraction* parser to read formulas and a corresponding *rendering* parser to render formulas according to the user-defined notation are generated. We sketch the basic mechanisms to accommodate efficiently modifications of the notations, to restrain ambiguities, to allow for the redefinition of notations and to use notations defined in other documents. Section 4 gives a worked example that illustrates the different facets of the support offered on that basis. We discuss related works in Section 5 before concluding in Section 6.

2 Preliminaries: ΩMEGA, MAYA, TASK LAYER and PLATΩ

The development of the proof assistance system ΩMEGA is one of the major attempts to build an all-encompassing assistance tool for the working mathematician or for the formal work of a software engineer. It is a representative of systems in the paradigm of *proof planning* and combines interactive and automated proof construction for domains with rich and well-structured mathematical knowledge. The ΩMEGA-system is currently under re-development where, among others, it has been augmented by the development graph manager MAYA, and the underlying natural deduction calculus has been replaced with the CORE-calculus [5].

The MAYA system [8] supports an evolutionary formal development by allowing users to specify and verify developments in a structured manner, it incorporates a uniform mech-

Figure 1. Architecture of integration of the text-editor and the ΩMEGA via the mediator PLATΩ

anism for verification in-the-large to exploit the structure of the specification, and it maintains the verification work already done when changing the specification. Proof assistance systems like ΩMEGA rely on mathematical knowledge formalized in structured theories of definitions, axioms and theorems. The MAYA system is the central component in the new ΩMEGA system that takes care about the management of change of these theories via its OMDOC-interface [13].

The CORE-calculus supports proof development directly at the *assertion level* [11], where proof steps are justified in terms of applications of definitions, axioms, theorems or hypotheses (collectively called *assertions*). It provides the logical basis for the so-called TASK LAYER [9], that is the central component for computer-based proof construction in ΩMEGA. The proof construction steps are: (1) the introduction of a proof sketch [23], (2) deep structural rules for weakening and decomposition of subformulas, (3) the application of a lemma that can be postulated on the fly, (4) the substitution of meta-variables, and (5) the application of an inference. Inferences are the basic reasoning steps of the TASK LAYER, and comprise assertion applications, proof planning methods or calls to external theorem provers or computer algebra systems (see [9,6] for more details about the TASK LAYER).

A formal proof requires to break down abstract proof steps to the CORE calculus level by replacing each abstract step by a sequence of calculus steps. This has usually the effect that a formal proof consists of many more steps than a corresponding informal proof of the same conjecture. Consequently, if we manually construct a formal proof many interaction steps are typically necessary. Formal proof sketches [23] in contrast allow the user to perform high-level reasoning steps without having to justify them immediately. The underlying idea is that the user writes down only the interesting parts of the proof and that the gaps between these steps are filled in later, ideally fully automatically (see also [19]). Proof sketches are thus a highly adequate means to realize the tight integration of a proof assistance system and a scientific text-editor.

The mediator PLATΩ [22] has been designed as a support system to realize the tight integration of a proof assistance system and a text-editor (see Figure 1). PLATΩ is connected with the text-editor by an informal representation language which flexibly supports the usual textual structure of mathematical documents. This semantic annotation language, called *proof language* (PL), allows for underspecification as well as alternative (sub)proof attempts. In order to generate the formal counterpart of a PL representation, PLATΩ sepa-

rates theory knowledge like definitions, axioms and theorems from proofs. The theories are formalized in the *development graph language* (DL), which is close to the OMDOC theory language supported by the MAYA system, whereas the proofs are transformed into the *tasklayer language* (TL) which are descriptions of TASK LAYER proofs. Hence, PLATΩ is connected with the proof assistance system ΩMEGA by a formal representation close to its internal datastructure.

Besides the transformation of complete documents, it is essential to be able to propagate small changes from an informal PL representation to the formal DL/TL one and the way back. If we always perform a global transformation, we would on the one hand rewrite the whole document in the text-editor which means to lose large parts of the natural language text written by the user. On the other hand we would reset the datastructure of the proof assistance system to the abstract level of proof sketches. For example, any already developed expansion towards calculus level or any computation result from external systems would be lost. Therefore, one of the most important aspects of PLATΩ's architecture is the propagation of changes.

The formal representation finally allows the underlying proof assistance system to support the user in various ways. PLATΩ provides the possibility to interact through context-sensitive service menus. If the user selects an object in the document, PLATΩ requests service actions from the proof assistance system regarding the formal counterparts of the selected object. Hence, the mediator needs to maintain the mapping between objects in the informal language PL and the formal languages DL and TL.

In particular, the proof assistance system supports the user by suggesting possible inference applications for a particular proof situation. Since the computation of all possible inferences may take a long time, a multi-level menu with the possibility of lazy evaluation is provided. PLATΩ supports the execution of nested actions inside a service menu which may result in a patch description for this menu.

Through service menus the user also gets access to automatic theorem provers and computer algebra systems which can be used to automatically verify conclusions and computations as well as suggest possible corrections. These and many more functionalities are supported by PLATΩ through its mechanism to propagate changes as well as the possibility of custom answers to the user of the text-editor.

A more detailed description of PLATΩ is given in [22]. Note that, generally we aim at an approach that is independent of the particular proof assistance system to be integrated. Therefore the proof language as well as the service menu language are parameterized over the sublanguages for definitions, formulas, references and menu argument content.

Presentational convention: The work presented in this paper has been realized in TEXMACS. Although the TEXMACS markup-language is analogous to LATEX-macros, one needs to get used to it: For instance a macro application like `\frac{A}{B}` in LATEX gets `<frac|A|B>` in TEXMACS-markup. Assuming that most readers are more familiar with LATEX than with TEXMACS, we will use a LATEX-syntax for the sake of readability.

3 Dynamic Notation

The PLATΩ system supports users to interact with a proof assistance system from inside the text-editor TEXMACS by offering service menus and by propagating changes of the doc-

Element	Arguments
\F	{*name*}{\B?, (\F |\V |\S)⋆ }
\B	{\V+ }
\V	{*name*, (\T |\TX |\TF)?}
\S	{*name*}
\T	{*name*}
\TX	{(\T |\TX |\TF), (\T |\TX |\TF) }
\TF	{(\T |\TX |\TF), (\T |\TX |\TF) }

Figure 2. Grammar of the annotation language for formulas.

ument to the system and vice versa. Mediating between a text-editor and a proof assistance system requires to extract the formal content of a document, which is already a challenge in itself if one wants to allow the author to write in natural language without any restrictions.

Therefore we currently use the semantic annotation language presented in [21] to semantically annotate different parts of a document. The annotations can be nested and subdivide the text into dependent theories that contain definitions, axioms, theorems and proofs, which themselves consist of proof steps like for example subgoal introduction, assumption or case split. The annotations are a set of macros predefined in a TEX$_{\text{MACS}}$ style file and must be provided manually by the author. We were particularly cautious that adding the annotations to a text does not impose any restrictions to the author about how to structure her text. Note that for the communication with the proof assistance system, also the formulas must be written in a fully annotated format whose grammar is shown in the table of Figure 2. `\F{name}{args}` represents the application of the function `name` to the given arguments `args`. `\B{vars}` specifies the variables `vars` that are bound by a quantifier. A variable `name` is denoted by `\V{name}` and may be optionally typed. A type `name` is represented by `\T{name}`. Complex types are composed using the function type constructor $\rightarrow$ represented by `\TF{type1, type2}`, or the pair type operator $\times$ represented by `\TX{type1,type2}`. Finally, a symbol `name` is denoted by `\S{name}`. For example, the formula $x \in A \cap (B \cup C)$ is represented by the fully annotated form `\F{in}{\V{x},\F{cap}{\V{A},F{cup}{\V{B},V{C}}}}`, the quantified formula $\forall x.\ x = x$ as `\F{forall}{\B{\V{x}},\F{=}{\V{x},\V{x}}}`. In many cases the type of a variable can be reconstructed using type inference or from bounded context variables, therefore typing variables is optional in our system.

The vision is on the one hand to combine our approach with the MATHLANG project [12] and on the other hand to use natural language analysis techniques for the semi-automatic annotation of the document structure, e.g. to automatically detect macro-structures like theories, definitions and theorems. Although these macro-structures might be annotated manually by the user at the moment, it is not acceptable to write formulas in fully annotated form. This motivates the need for an *abstraction* parser that converts formulas in LaTeX syntax into their fully annotated form. Furthermore, we also need a *rendering* parser to convert fully annotated formulas obtained from the proof assistance system into LaTeX-formulas according to the user-defined notation.

The usual software engineering approach would be to write grammars for both directions and integrate the generated parsers into the system. Of course, this method is highly efficient but the major drawback is obvious: the user has to maintain the grammar files together with her documents. In our document-centric philosophy, the only source of

knowledge for the mediator and the proof assistance system should be the document in the text-editor. Therefore, instead of maintaining special grammar files for the parser, the idea of dynamic notation is to synthesize these grammar informations from the definitions and notations occurring in the same document where they are used. This WYSIWYG style of defining notation starts from basic abstraction and rendering grammars for types and formulas, where only the base type *bool*, the complex type constructors $\rightarrow, \times$ [1] and the logic operators $\forall, \exists, \lambda, \top, \bot, \wedge, \vee, \neg, \Rightarrow, \Leftrightarrow$ are predefined. This set of basic types and logic operators is a parameter of the PLATΩ system that has to be specified for each proof assistant system. A component of PLATΩ maps them into the input syntax of the particular proof assistant (e.g. Π, o) and the way back. Since mathematicians in general are not familiar with the type *o*, we decided to use the name *bool* instead. Based on that initial grammar the definitions and notations occurring in the document are analyzed in order to extend incrementally both grammars for dealing with new symbols, types and operators. The scope of a notation should thereby respect the visibility of its defining symbol or type, i.e. the transitive closure of dependent theories. Finally, all formulas are parsed using their theory specific *abstraction* parser.

Notations defined by authors are typically not specified as grammar rules. Therefore, we first need a user friendly WYSIWYG method to define notations and to automatically generate grammar rules from it. Looking at standard mathematical textbooks, one observes sentences like *"Let x be an element and A be a set, then we write $x \in A$, x is element of A, x is in A or A contains x."*. Supporting this format requires the ability to locally introduce the variables *x* and *A* in order to generate grammar rules from a notation pattern like $x \in A$. Without using a linguistic database, patterns like *x is in A* are only supported as pseudo natural language. Beside that, the author should be able to declare a symbol to be right- or left-associative as well as precedences of symbols.

We introduce the following annotation format to define the operator $\in$ and to introduce multiple alternative notations for $\in$ as closely as possible to the textbook style.

```
\begin{definition}{Function $\in$}
  The predicate \concept{\in}{elem \times set \rightarrow bool}
  takes an individual and a set and tells whether that
  individual belongs to this set.
\end{definition}

\begin{notation}{Function $\in$}
  Let \declare{x} be an individual and \declare{A} a set,
  then we write \denote{x \in A}, \denote{x is element of A},
  \denote{x is in A} or \denote{A contains x}.
\end{notation}
```

Figure 3 shows how the above example definition and notation appear in a TEXMACS document. A definition may introduce a new type by `\type{name}` or a new typed symbol by `\concept{name}{type}`. We allow to group symbols to simplify the definition of precedences and associativity. By writing `\group{name}` inside the definition of a symbol, this particular symbol is added to the group `name` which is automatically created if

[1] Here $\times$ is not a pair type constructor, but only syntactic sugar that allows to write $\tau_1 \times \tau_2 \rightarrow \tau_0$ as short-hand for $\tau_1 \rightarrow (\tau_2 \rightarrow \tau_0)$.

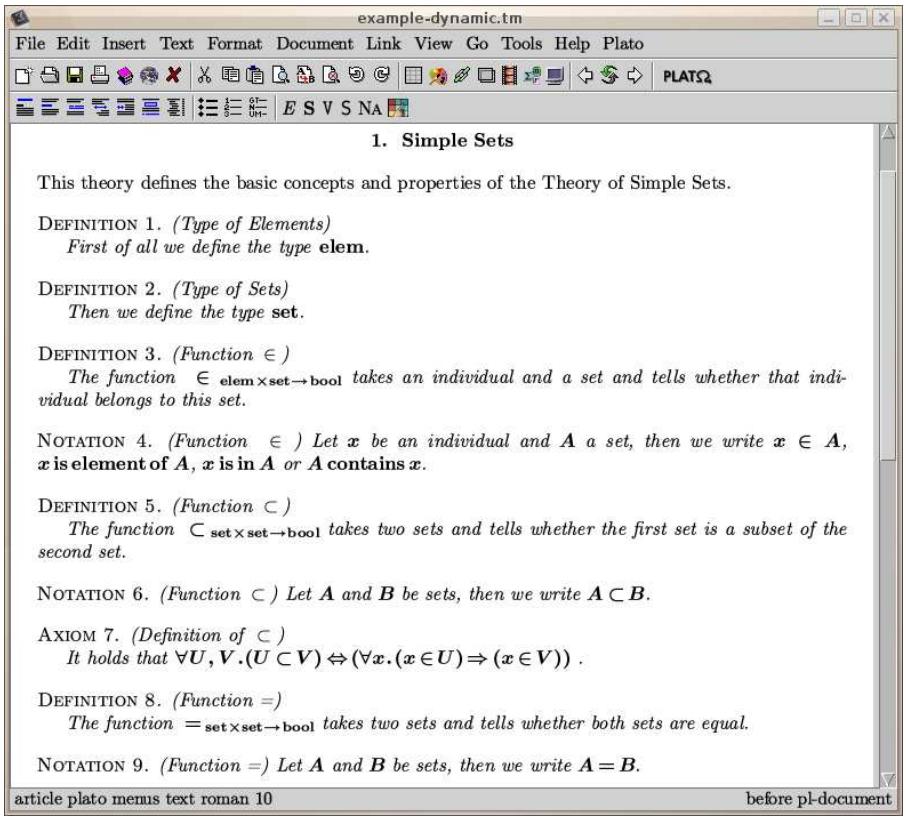

Figure 3. TEXMACS document with dynamic notation

it does not exist. A notation may contain some variables declared by `\declare{name}` as well as the patterns written as `\denote{pattern}`. Furthermore, one can specify a symbol or group of symbols to be left- or right-associative by writing `\left{name}` or `\right{name}` inside the notation. Finally, precedences between symbols or groups are defined by `\prec{name1,...,nameN}`, which partially orders the precedence of these symbols and groups of symbols from low to high. Declarations are rejected if conflicting information is detected during the computation of the transitive closure. Information about association and precedence is considered to be visible in the whole theory whereas the visibility of notation starts at the position of its declaration. Please note that a notation is related to a specific definition by referring its name, in our example `Function $\in$`.

Starting the processing of a semantically annotated document, as for example the document shown in Figure 3, all surrounding natural language parts in the document are removed and the *abstraction* and *rendering* parsers and scanners are initialized with the initial grammars for types and formulas. The goal of processing the document is to produce a set of grammar rules for the respective grammars from the notational definitions given in the text. A top-down approach, that processes each definition or notation on its own, extending the grammars and recompiling the parsers before processing the next element, is far too inefficient for real time usage due to the expensive parser generation process [2] . The diagram shown in Figure 4 describes the procedure that we use to minimize the amount of parser generations as much as possible.

[2] The parser generator is implemented and compiled in Lisp to be part of PLATΩ. For instance, the parser generation for the example document takes $\approx 3sec$.

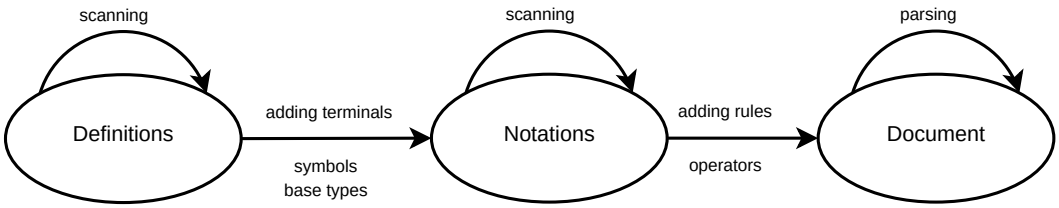

Figure 4. Procedure to minimize the amount of parser generations

- Phase 1: All definitions are processed sequentially. For each definition the name of the introduced type or symbol is added to both grammars.
- Phase 2: All notations are processed sequentially. For each notation the introduced patterns are analyzed to generate rules for both grammars.
- Phase 3: *Abstraction* and *rendering* parsers are rebuilt and all formulas are processed.

The definitions and notations are analyzed to generate grammar rules for the *abstraction* grammar; in case different notations are defined for a symbol, this results in multiple grammar rules. For the *rendering* grammar we currently select the first given notation. The parser generator then creates the respective *abstraction* and *rendering* parsers. The details of the analysis and the parser creation can be found in [7].

3.1 Management of Change for Notations

The constructed *abstraction* parser is for one version of the document. When the author continues to edit the document, it may be modified in arbitrary ways, including the change of existing definitions and notations. Before the modified semantic content of the document is uploaded into the proof assistance system, we need to recompute the parsers and parse the formulas in the document. Always starting from scratch following the procedure described in the previous section is highly inefficient and may jeopardize the acceptance by the author of the system if that process takes too long. Therefore there is a need for management of change for the notational parts of a document and those parts that depend on them. The management of change task has three aspects:

- First, we must determine any modifications in the notational parts: we re-process all definitions and notations of the document and obtain a new set of grammar rules. By caching these sets of rules for each symbols, we can determine the grammar changes.
- Second, we must adjust only those parts of the grammar that are affected by the determined modifications and adjust the parsers accordingly. If the modification of the grammar is non-monotonic, i.e. some rules have been removed or changed, we currently have to recompute the whole parser from scratch. This is for instance the case if we change a notation for some symbol, e.g. replacing *"$A \subset B$"* by *"$B \supset A$"*, but not if we add an additional alternative notation for a symbol, like allowing *"A is subset of B"* in addition to *"$A \subset B$"*. If the grammar is simply extended, we can optimize the creation of the new parser (see [7] for details).
- Third, we must re-parse (resp. re-render) the formulas of the document that are affected by the changes. Once the parser has been adjusted we need to re-parse those formulas the author has written or changed manually, e.g. if the formula in the axiom of Figure 3 would have been replaced by $\forall U,V.\ (U\ \mathit{is\ subset\ of}\ V) \Leftrightarrow (\forall x.\ (x\ \mathit{is\ in}\ U) \Rightarrow$

$(V \textit{ contains } x)$). Furthermore, we have to re-render those formulas that were generated by the proof assistance system. To this end we store the following information on formulas in the document: for each formula we have a flag indicating if it was generated by the proof assistance system, the corresponding fully annotated formula, and the set of grammar rules that were used for parsing (resp. rendering) that formula:

- If the formula was written by the author, the associated fully annotated formula and the grammar rules are the result of the *abstraction* parser.
- If a formula was generated by the proof assistance system, that formula is the result of *rendering* the fully annotated formula obtained from the proof assistance system. The stored grammar rules are those returned by the *rendering* parser.

Note that we do not prevent the author from editing a generated formula. As soon as the author edits such a formula, the flag attached to the formula is toggled to "user" and the cached fully annotated version and grammar rules are replaced during the next *abstraction* parsing of the formula. The stored information is used to optimize the next parsing (resp. rendering) pass over the document: A formula is only parsed from scratch if at least one of the used grammar rules has been modified or deleted, or if either the user or the proof assistance system has changed the formula (resp. the fully annotated formula).

3.2 Ambiguities, Dependencies and Libraries

In order to enable a document-centric approach for formalizing mathematics and software, the added-values offered by the authoring environment must outweigh the additional burden imposed to the author compared to the amount of work for a non-assisted preparation of a document. We exploit the theory structure contained in a document to reduce the ambiguities the author would have to deal with and also support the redefinition of notations. With respect to added-values, we provide checks if a notation is used before it has been introduced in a document and, most importantly, we support an author to build on formalizations contained in other documents.

Ambiguities. The *abstraction* parser returns all possible readings and during the parsing process can try to make use of type-reconstruction provided by a so-called *refiner* [18] to eliminate possible readings that are not type-correct. This uses the logical context of a formula which is determined by the theory it occurs in: the different parts of a document must be assigned to specific theories. New theories can be defined inside a document and built on top of other theories. The notion of theory is that of OMDOC [13] respectively development graphs [15]. The other possibility consists of avoiding some ambiguities that would arise when sticking to have a single parser by having different parsers for different theories. As an example consider a theory of the integers with multiplication with the notation "$x \times y$" and a completely unrelated theory about sets and Cartesian products with the same notation. This typically is a source of ambiguities that would require the use of type information to resolve the issue. Using different parsers for different theories completely avoids that problem.

Redefining Notations. When importing a theory, we want to reuse the formal content, but possibly adapt the notation used to write formulas. This occurs less frequently inside

a single document, but occurs very often when using a theory formalized in a different document. Since we linked the parsing (and hence the rendering) to the individual theories, we allow to redefine notations for symbols inherited from other theories. The grammar rules for a parser are determined by including for each imported symbol that notation which is closest in the import hierarchy of theories. If there are two such theories [3], a conflict is raised and the author must interactively advise which notation to use.

Dependencies. A parser and the associated renderer are attached to a theory and each position in the document belongs to a theory. Therefore, it is possible that within a specific theory, a formula uses the notation of some symbol although the definition of that notation only occurs afterwards in the document. We notice such situations by comparing the position in the document where grammar rules are defined and where they have been used to parse a formula [4]. If we determine such a situation, we could try to re-arrange the document by inspecting the grammar rules used for *abstraction* and *rendering*. However, our impression is that most authors would be upset if their document is rearranged automatically. Therefore, we only notify the author in these situations.

Libraries. A library mechanism is the key prerequisite to support the development of large structured theories. We carry over that concept to the document-centric approach we are aiming at by extending the citation mechanism that is commonly used in documents. PLATΩ provides a macro to cite a document *semantically*, i.e. it will not only be included in the normal bibliography of the document, but the formalized content of the document is included. Aside from the extracted formalizations we extract the notations contained in that document. The structured theory approach for parsers turns out to be again particularly beneficial to support that process and to redefine the notations for imported symbols.

4 A Worked Example

In this section we will illustrate the integration by discussing a worked example in the theory of Simple Sets. The mediation between the informal representation in the text-editor and the formal representation in the proof assistance system will be described on an abstract level. All details on the communicated documents, patch descriptions and menus for this example can be found in [21].

Since the TEXMACS interface for proof assistance systems is under continuous development, a PLATΩ plugin for TEXMACS has been developed by the ΩMEGA group that maps the interface functions of PLATΩ to the current ones of TEXMACS and which defines a style file for PL macros in TEXMACS. In the following example, we use this plugin to establish a connection between TEXMACS and PLATΩ's XML-RPC server.

In the text-editor, we have written an example document with the semantic annotation language PL (defined in [21]). The theory *Simple Sets* in this document contains for example definitions and axioms for $\subset$, $=$, $\cup$ and $\cap$. Figure 3 (p. 7) and the top of Figure 5 (p. 11) show the theory as seen in TEXMACS. Furthermore, we have written a theory *Distributivity in Simple Sets* which imports all knowledge from the first theory *Simple Sets*. This second theory consists of a theorem about the *Distributivity of* $\cap$. The user has already started a

[3] The theories can form an acyclic graph which may lead to the Nixon diamond scenario.

[4] A similar dependency is between a definition of a concept and the definition of its notation.

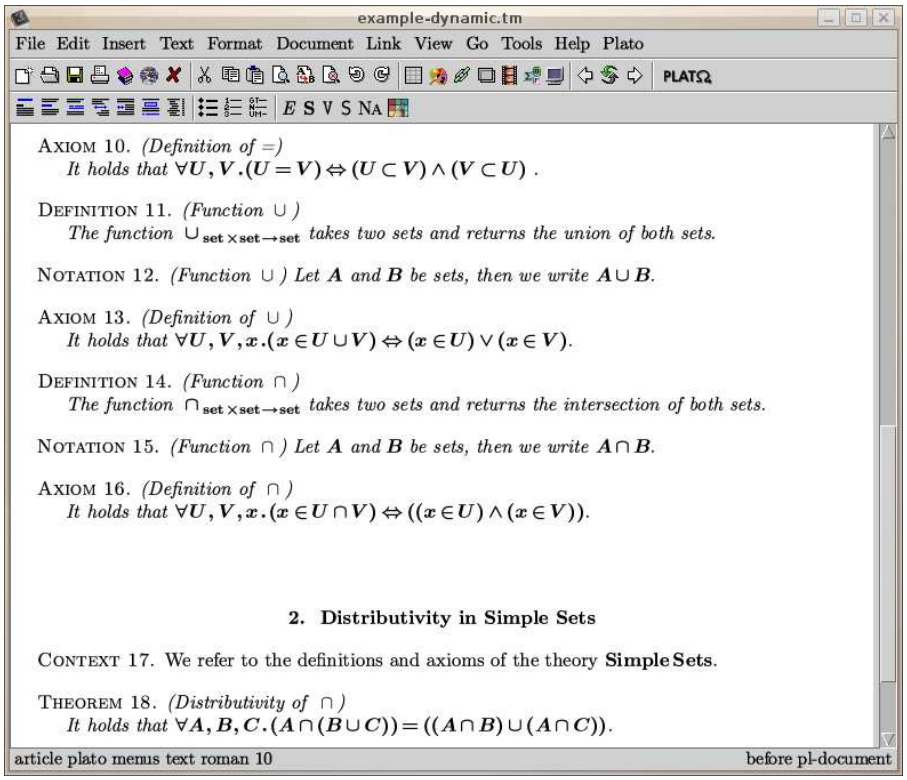

Figure 5. Theories *Simple Sets* and *Distributivity in Simple Sets* in TEXMACS

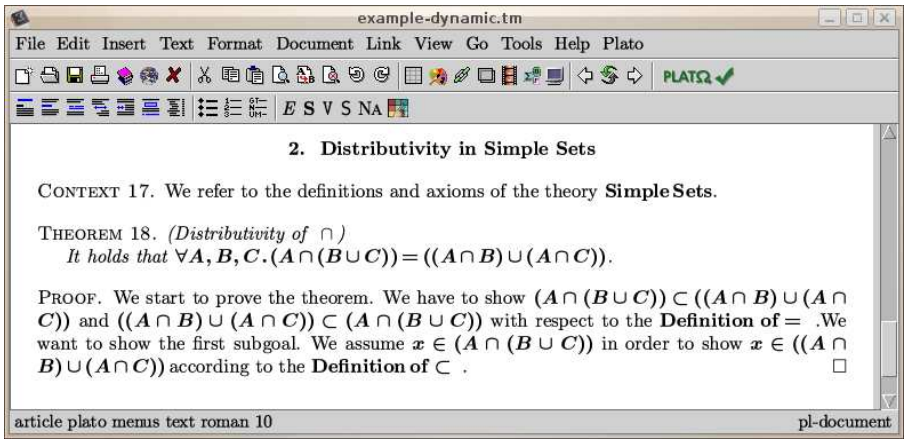

Figure 6. Manually written partial proof in TEXMACS

proof for this theorem by introducing two subgoals. Figure 6 shows the theory as seen in TEXMACS.

The PL macros contained in the document must be provided by the user and are used to automatically extract the corresponding PL document, the informal representation of the document for PLATΩ. Uploading this PL document, PLATΩ creates a DL document containing definitions, axioms and theorems in a representation close to OMDOC that MAYA takes as input for the creation of a development graph. The partial proof is transformed into a TL document, an abstract representation for the TASK LAYER.

Further developing the document, the user has started to prove the first subgoal by deriv-

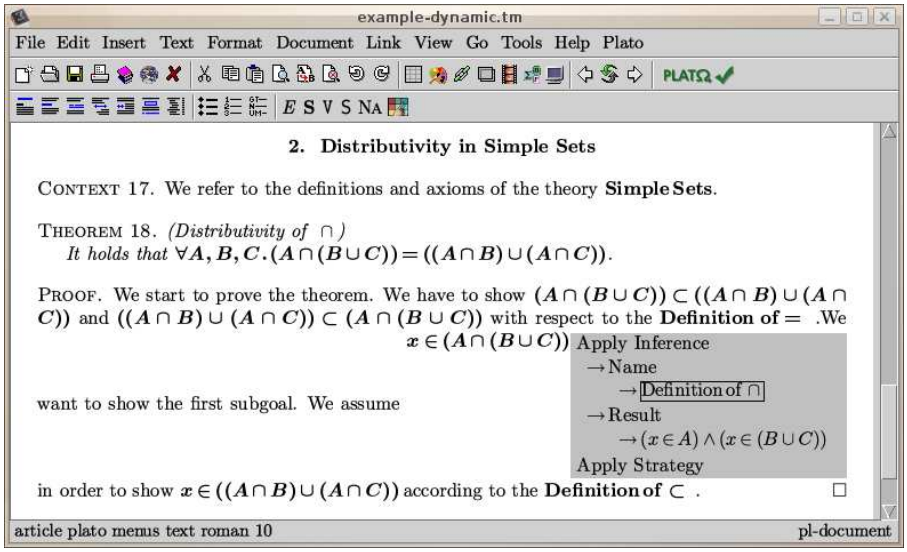

Figure 7. Interactive Proof Construction via Service Menus

ing a new subgoal and introducing an assumption (see Figure 6). In general, the difference with respect to the last synchronized version of the document should be computed and sent to PLATΩ. At the moment, TEXMACS is not able to compute these differences, therefore the whole document is uploaded again and PLATΩ computes the differences in the form of a patch. That patch for the informal PL document is then transformed by PLATΩ to a patch for the formal representations in DL and TL. In our example, the modifications do not affect the theory knowledge and the transformation only results in modifications for the representation of the TASK LAYER. Altogether, the user is able to synchronize her informal representation in the text-editor document with the formal representation in the proof assistance system.

The next interesting feature of PLATΩ is the possibility of getting system support from the underlying proof assistance system. Selecting the recently introduced formula in the assumption, the user requests a service menu from PLATΩ. Requesting services for the corresponding task in the TASK LAYER, a list of available inferences is returned to PLATΩ. In order to answer quickly to the text-editor, we generate nested actions that allow to incrementally compute the formulas resulting from the application of an inference rather than to precompute all possible resulting formulas for all available inferences. Note that the inferences are automatically generated from the axioms, lemmas and theorems contained in the document.

The menu is displayed to the user in TEXMACS, who can select which inference to apply. The selection triggers the computation of all resulting formulas, for instance for the inference **Definition of** ∩, defined by the corresponding axiom. PLATΩ tells the text-editor how to change the menu by sending a patch description for the menu. The user selects the desired formula (see Figure 7) which triggers the application of the inference. PLATΩ calls the TASK LAYER for the application of the selected inference in order to obtain the chosen formula. The TASK LAYER performs the requested operation which typically modifies the proof of TASK LAYER. This modification is transformed by PLATΩ into a patch description for the formal representation in TL and subsequently into a PL patch for the informal document in TEXMACS, which is then sent to the text-editor. Furthermore, the menu is

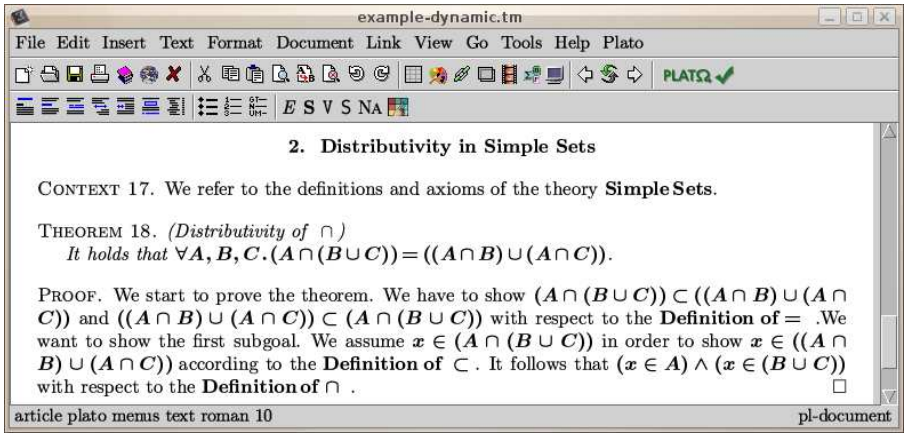

Figure 8. Modification of the Proof in TEXMACS by PLATΩ

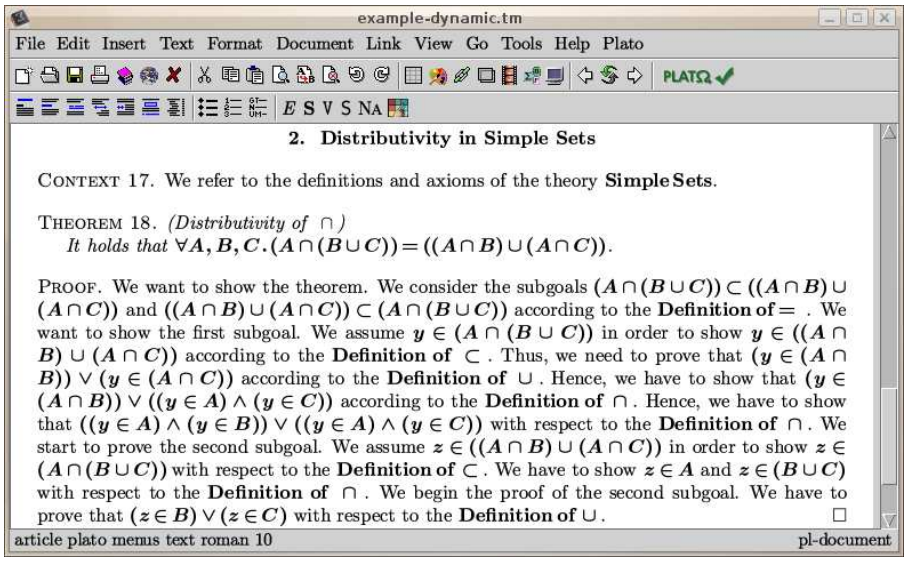

Figure 9. Automatically Constructed Proof in TEXMACS.

closed by sending a patch description which removes it. Currently, the new proof fragments are inserted using predefined natural language fragments to describe the proof steps and the current rendering parser is used to render the formula in the notation defined by the author. Future work will consist of integrating the natural language proof presentation system P.REX [10] into PLATΩ, in order to generate true natural language output for the proof steps added by the proof assistance system. The text-editor finally patches the document according to this patch description. The resulting document is shown in Figure 8.

Aside from interactive proof construction, the author can call a proof procedure that searches for a proof on the TASK LAYER. In this case a sequence of proof steps is added to the TASK LAYER which are then transformed into patches for PLATΩ. Figure 9 shows such a proof in TEXMACS which uses the rendering parser to present the new formulas.

Note that the user can change any part of the document, including the parts generated by the proof assistance system. Due to the maintenance of consistent versions, the further development of the document can be a mix of manual authoring by the user and interactive authoring with the proof assistance system.

5 Related Work

To our knowledge there has not been any attempt to integrate a proof assistance system with text-editors in the tight and flexible way as done via PLATΩ. Approaches like AUTOMATH, MIZAR, ISAR, and THEOREMA do not consider the input document as an independent, first-class citizen with an internal state that has to be kept consistent with the formal representations in the proof assistance system while allowing arbitrary changes on each side. The first attempts in that direction have been carried out in the context of the MATITA prototype which used to have an integrated component to re-annotate computer generated representation of proofs. Since re-generation was a slow non-incremental operation based on complex XSLT style-sheets, re-annotating the text was an unpleasant experience for the user, which hampered user-acceptance of that part of the MATITA system. A similar attempt in that direction has been carried in the context of PROOF GENERAL [3]. In PROOF GENERAL the user edits a central document in a suitable editing environment, from which it can be evaluated by various tools, such as a proof assistance system, which checks whether the document contains valid proofs, or a renderer which typesets or renders the document into a human-oriented documentation readable outside the system. However, the system is only an interface to proof assistance systems that process their input incrementally. Hence, the documents edited in PROOF GENERAL are processed incrementally in a top-down manner, parts that have been processed by the proof assistance system are locked and cannot be edited by the user. Furthermore, the documents are in the input format of the proof assistance systems rather than in the format of some type-setting program. Though we have tried to design the functionalities and representation languages in PLATΩ's interface as general as possible, future work will have to show that PLATΩ can be easily adapted to different proof assistance systems (like PROOF GENERAL).

With respect to supporting the definition of new notations that are used for type-setting, the systems ISABELLE [16] and MATITA [2] are closest. ISABELLE comes with type-setting facilities of formulas and proofs for LaTeX and supports the declaration of the notation for symbols as prefix, infix, postfix and mix-fix. Furthermore, it allows the definition of *translations* which are close to our style of defining notations. The main differences are: ISABELLE processes the input document containing the declaration of the notation with a top-down mechanism and finally generates an output document in LaTeX. Since in our WYSIWYG setting, input and output document are physically the same document, the document is processed in multiple phases: first the definition of types and symbols are identified, second the parser rules are synthesized from the declarations of notation, third the document is processed with a family of local parsers. Due to the batch processing paradigm of ISABELLE, there are no mechanisms to efficiently deal with modifications of the notation, which is crucial in our interactive authoring environment.

In the context of MATITA Padovani and Zacchiroli also proposed a mechanism of *abstraction* and *rendering* parsers [17] that are created from notational equations which are comparable to the grammar rules we generate from the notational definitions. Their mechanism is mainly devoted to obtain MathML representations [1] where a major concern also is to maintain links between the objects in MathML to the internal objects. Similar to ISABELLE, the input document is processed top-down and is separated from the output document in MathML. Also, they do not consider the effect of changing the notations and to efficiently adjust the parsers. The main difference to the approaches in ISABELLE and

MATITA is that in our approach the parsers and renderers are part of the user interface and not of the proof assistance system.

Audebaud and Rideau presented in the context of the COQ proof assistant a command line interface connection with TEXMACS called TMCOQ [4]. Furthermore Mamane and Geuvers are developing a more document-centric proof script interface called TMEGG [14] which uses similar techniques as PROOF GENERAL with respect to the batch processing of the document.

6 Conclusion

In order to enable a document-centric approach for formalizing mathematics and software, the added-values offered in an assisted authoring environment must outweigh the additional burden imposed to the author compared to the amount of work for a non-assisted preparation of a document. We presented the mediator PLATΩ that allows the user to write her mathematical documents in the language she is used to, that is a mixture of natural language and formulas, and to define and use her own notation inside a document.

Building theory-specific parsers from these notation definitions, PLATΩ automatically builds up the corresponding formal representation from formulas written by the user in the LATEX-style she is used to and renders the formulas produced by the proof assistance system. It provides a sophisticated management of change for notations and takes further care of the maintenance of consistent versions in TEXMACS and the proof assistance system. All kinds of services of the underlying proof assistance system regarding the formal representation can be provided inside the text-editor through PLATΩ as context-sensitive service menus.

This paper illustrates the major aspects that are supported in the current version of the integration. Further work will consist of reducing the amount of annotations the user has to provide by employing natural language analysis techniques. Finally, we plan to increase the interoperability with other proof assistant systems as well as tutorial dialogue systems.

References

[1] Mathematical Markup Language (MathML) Version 2.0. W3c recommendation 21 february 2001. Technical report, `http://www.w3.org/TR/MathML2`, 2003.

[2] A.Asperti, C. Sacerdoti-Coen, E.Tassi, and S.Zacchiroli. User interaction with the Matita proof assistant. *Journal of Automated Reasoning, Special Issue on User Interfaces for Theorem Proving*, 2007. To appear.

[3] D. Aspinall, C. Lüth, and B. Wolff. Assisted proof document authoring. In M. Kohlhase, editor, *Mathematical Knowledge Management MKM 2005*, volume 3863 of *LNAI*, pages 65– 80. Springer, 2006.

[4] P. Audebaud and L. Rideau. TEXMACS as authoring tool for publication and dissemination of formal developments. In D. Aspinall and C. Lüth, editors, *5th Workshop on User Interfaces for Theorem Provers (UITP'03)*, ENTCS. Elsevier, September 2003.

[5] S. Autexier. The CORE calculus. In R. Nieuwenhuis, editor, *Proceedings of CADE-20*, LNAI 3632, Tallinn, Estonia, July 2005. Springer.

[6] S. Autexier and D. Dietrich. Synthesizing proof planning methods and Ωants agents from mathematical knowledge. In J. Borwein and B. Farmer, editors, *Proceedings of MKM'06*, volume 4108 of *LNAI*, pages 94–109. Springer, 2006.

[7] S. Autexier, A. Fiedler, T. Neumann, and M. Wagner. Supporting user-defined notations when integrating scientific text-editors with proof assistance systems. In M. Kerber and R. Miner, editors, *Proceedings of MKM'07*. Springer, 2007.

[8] S. Autexier and D. Hutter. Formal software development in Maya. In D. Hutter and W. Stephan, editors, *Festschrift in Honor of J. Siekmann*, volume 2605 of *LNAI*. Springer, February 2005.

[9] D. Dietrich. The Task Layer of the ΩMEGA System. Diploma thesis, Saarland University, Saarbrücken, Germany, 2006.

[10] A. Fiedler. *User-adaptive Proof Explanation*. Phd thesis, Naturwissenschaftlich-Technische Fakultät I, Universität des Saarlandes, Saarbrücken, Germany, 2001.

[11] X. Huang. *Human Oriented Proof Presentation: A Reconstructive Approach*. Number 112 in DISKI. Infix, Sankt Augustin, Germany, 1996.

[12] F. Kamareddine, M. Maarek, and J. B. Wells. Toward an object-oriented structure for mathematical text. In M. Kohlhase, editor, *MKM 2005, Fourth International Conference on Mathematical Knowledge Management*, LNAI 3863, pages 217–233. Springer, 2006.

[13] M. Kohlhase. *OMDOC - An Open Markup Format for Mathematical Documents [Version 1.2]*, volume 4180 of *LNAI*. Springer, August 2006.

[14] L. E. Mamane and H. Geuvers. A document-oriented Coq plugin for TEXMACS. In M. Kerber and R. Miner, editors, *Proceedings of MKM'07*. Springer, June 2007.

[15] T. Mossakowski, S. Autexier, and D. Hutter. Development graphs - proof management for structured specifications. *Journal of Logic and Algebraic Programming, special issue on Algebraic Specification and Development Techniques*, 67(1-2):114–145, April 2006.

[16] T. Nipkow, L. C. Paulson, and M. Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS*. Springer, 2.

[17] L. Padovani and S. Zacchiroli. From notation to semantics: There and back again! In J. Borwein and B. Farmer, editors, *Proceedings of MKM'06*, volume 4108 of *LNAI*. Springer, August 2006.

[18] C. Sacerdoti-Coen and S. Zacchiroli. Efficient ambiguous parsing of mathematical formulae. In A. Asperti, G. Bancerek, and A. Trybulec, editors, *Proceedings of the Third International Conference on Mathematical Knowledge Management*, volume 3119 of *LNCS*, pages 347–362, Bialystok, Poland, September 2004. Springer.

[19] J. Siekmann, C. Benzmüller, A. Fiedler, A. Meier, and M. Pollet. Proof development with OMEGA: $\sqrt{2}$ is irrational. In M. Baaz and A. Voronkov, editors, *Logic for Programming, Artificial Intelligence, and Reasoning, 9th International Conference, LPAR 2002*, number 2514 in LNAI, pages 367–387. Springer, 2002.

[20] J. van der Hoeven. Gnu TEXMACS: A free, structured, WYSIWYG and technical text editor. Number 39-40 in Cahiers GUTenberg, May 2001.

[21] M. Wagner. Mediation between text-editors and proof assistance systems. Diploma thesis, Saarland University, Saarbrücken, Germany, 2006.

[22] M. Wagner, S. Autexier, and C. Benzmüller. PLATΩ: A mediator between text-editors and proof assistance systems. In C. Benzmüller S. Autexier, editor, *7th Workshop on User Interfaces for Theorem Provers (UITP'06)*, ENTCS. Elsevier, August 2006.

[23] F. Wiedijk. Formal proof sketches. In S. Berardi, M. Coppo, and F. Damiani, editors, *Types for Proofs and Programs: Third International Workshop, TYPES 2003*, LNCS 3085, pages 378–393, Torino, Italy, 2004. Springer.