Organization, Transformation, and Propagation of Mathematical Knowledge in Ωmega

Serge Autexier, Christoph Benzmüller, Dominik Dietrich and Marc Wagner

Abstract. Mathematical assistance systems and proof assistance systems in general have traditionally been developed as large, monolithic systems which are often hard to maintain and extend. In this article we propose a *component network architecture* as a means to design and implement such systems. Under this view a mathematical assistance system is an integrated knowledge-based system composed as a network of individual, specialized components. These components manipulate and mutually exchange different kinds of mathematical knowledge encoded within different document formats. Consequently, several units of mathematical knowledge coexist throughout the system within these components and this knowledge changes non-monotonically over time. Our approach has resulted in a lean and maintainable system code and makes the system open for extensions. Moreover, it naturally decomposes the global and complex reasoning and truth maintenance task into local reasoning and truth maintenance tasks inside the system components. The interplay between neighboring components in the network is thereby realized by non-monotonic updates over agreed interface representations encoding different kinds of mathematical knowledge.

1. Introduction

The long-term goal of the ΩMEGA project is the development of a large, integrated assistance system supporting different mathematical tasks and a wide range of typical research, publication and knowledge management activities. Examples of such

tasks and activities are computing, proving, solving, modeling, verifying, structuring, maintaining, searching, inventing, paper writing, explaining, illustrating, and possibly others.

The engineering of such a system is a non-trivial task for several reasons:

- Mathematical assistance systems are knowledge-based systems in which different kinds of knowledge must be maintained. This includes pure logical knowledge like axioms and lemmas organized in structured theories, partial proofs, proof procedures, procedure specific information like an ordering of constant symbols or proof planning methods, and notational information used in the user-interfaces of the system. Furthermore, the same piece of knowledge may exist in different, task specific representations. For example, a declaratively represented axiom in some mathematical theory may correspond to an operationally available inference or rewrite rule in the system's proof procedures.
- Mathematical knowledge development is an evolutionary, non-monotonic process. New knowledge is defined but also existing knowledge is removed or transformed, which may in turn affect already derived knowledge. Simply deleting derived knowledge and trying to re-derive it is often not a suitable option, in particular, if the derivation has been costly in the first place.
- Large mathematical assistance systems are typically developed by several people in parallel, each working on different aspects and adding new functionalities. It is thus a challenge to keep the system maintainable and at the same time open for quick and easy inclusions of new functionalities. Typically such new functionalities come together with new forms of mathematical knowledge to be maintained and with dependencies to already existing knowledge in the system. As we have learned from our own experience with the former ΩMEGA system [7], a monolithic system design that mixes all knowledge forms and dependencies in a single representation not only increases the complexity of the system by creating opaque dependencies, it also quickly makes the whole system resistant for any further extension and increasingly hard to maintain.

In this paper we propose a *component network architecture* as a means to design and implement mathematical assistance systems. The key idea is to achieve a clean separation of competency by explicitly defining and implementing components that are concerned with specific tasks (and related knowledge dependencies) only. Each component exchanges knowledge with neighboring components by non-monotonic updates over agreed interface representations encoding different kinds of mathematical knowledge. This results in an overall system that (i) supports the evolution of mathematical knowledge, (ii) has lean internal representations in each component, (iii) avoids accidental and spurious dependencies, and therefore (iv) remains maintainable while being open for extensions.
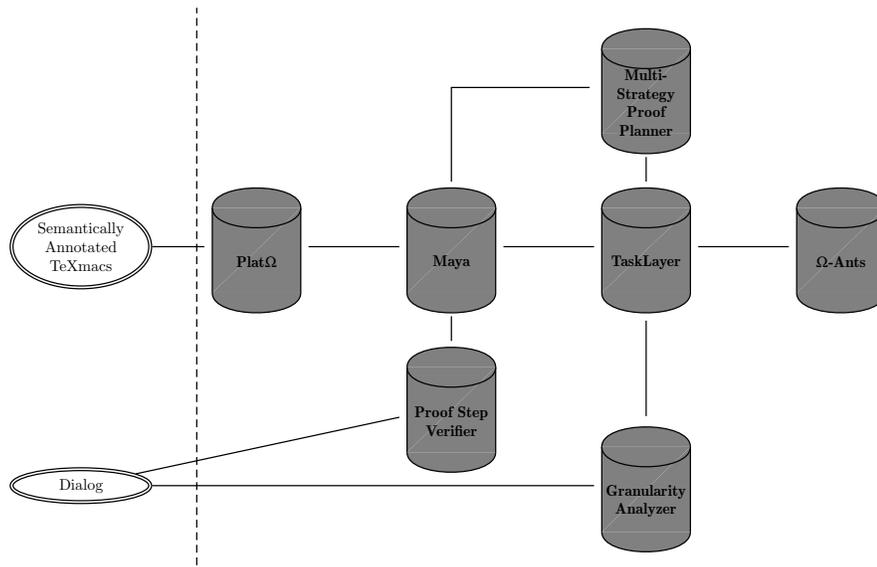
Figure 1: Networked components of the proof assistance system ΩMEGA

The paper is organized as follows: In Section 2 we describe ΩMEGA's main components [24, 23] and motivate the component network architecture by a running example. In Section 3 we describe the representations and update operations of the interfaces between the different components. We then illustrate our component network by presenting in detail the propagation of updates through the network in the different phases of the running example. In Section 4 we discuss the benefits of our architecture for current and future extensions of the ΩMEGA system. We compare the approach to related work in Section 5 and conclude the paper in Section 6.

2. Component based Architecture of Ωmega

The organization of the ΩMEGA system as a component network is presented in Figure 1: The network consists of components ▯ of the ΩMEGA system and external systems ⊚. In this Section we will introduce the components, illustrate their interplay with the help of an example, and sketch the overall realization of ΩMEGA as a component network.

2.1. The Components

The two core components of ΩMEGA are MAYA and the TASKLAYER.

Maya – The Theory Manager: Mathematical domain knowledge such as axioms, definitions, lemmas, and theorems – collectively called *assertions* – are organized in axiomatic theories and dynamically maintained in MAYA. These theories are built on top of each other by importing knowledge from lower theories via theory morphisms. MAYA's internal organization is based on the notion of *development graphs* [18]. Development graphs are exploited by MAYA to dynamically control which knowledge is available for which conjecture.

Each theory in a development graph contains standard information like the signature of its mathematical concepts and corresponding axioms, lemmas and theorems. In addition to these notions, a development graph supports the maintenance of other kinds of knowledge. This additional knowledge not necessarily affects the semantics of a theory but may provide, for example, valuable information for proof procedures. An example is ordering information for the function symbols in the signature of a theory, which can be exploited by simplification procedures. Other examples are *inferences* (the proof construction operators of the TASKLAYER), *control rules*, and *strategies*.

Finally, MAYA maintains the information on how global proof obligations are decomposed into local proof obligations and which axioms, lemmas and theorems have been used in which proofs. These dependency relations are exploited to provide an efficient support for basic update operations to change the graph structure of the theories (for example, adding or removing new theories and import relations) as well as their content (for example, adding or deleting axioms or conjectures). Further information on MAYA is available in [5].

TaskLayer – The Proof Manager: The TASKLAYER provides the central means to represent, manipulate and maintain proofs in the ΩMEGA system and it offers a rich set of support functionalities for proof construction. Its main tasks are:

 (i) to maintain the current states of proof attempts with their open goals, and
 (ii) to offer interactive services for proof construction.

To accomplish task (i), the TASKLAYER collaborates with MAYA, which maintains the mathematical theories, including conjectures to be proved and their current proof status as well as dependencies between these theories.

For task (ii), the TASKLAYER provides the notion of an *inference* as the basic unit for proof construction. Inferences are the basic proof construction operators used by the multiple strategy proof planner MULTI and the agent-based suggestion component Ω-ANTS. They are either operational representations of domain axioms, lemmas and theorems or they encode user-defined, domain or problem specific mathematical methods, possibly including the use of specialized computing and reasoning systems.

The basic mathematical entities maintained by the TASKLAYER are so-called (proof) *tasks*, which are Gentzen-style multi-conclusion sequents [13], augmented by means to define multiple foci of attention on subformulas that are maintained during the proof. Each task is reduced to a possibly empty set of subtasks by one of the following proof construction steps: (1) the introduction of a proof sketch [3, 28],

(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. An eminent feature of the TASKLAYER is that inferences cannot only be applied on top-level formulas in a task, but also to subformulas. The operationalization of mathematical knowledge into inferences paired with the possibility to apply inferences to subformulas results in natural, human-oriented proofs where each step is justified by a mathematical fact, such as a definition, an axiom, a theorem or a lemma.

The TASKLAYER supports the representation of alternative proof steps for *both* the (horizontal) reduction of a goal as well as for the (vertical) expansion of a complex proof step to higher granularity. Further information on the TASKLAYER can be found in [12].

The TASKLAYER is connected to the proof planner MULTI and the reactive agent-based reasoning system Ω-ANTS. MULTI supports automated proof construction and Ω-ANTS supports the dynamic generation of suggestions (to the user or other systems) on how to continue a partial proof.

Multi – The Proof Planner: The multi-strategy proof planner MULTI performs a heuristically guided search using the proof strategies which it dynamically receives from MAYA. A proof strategy describes a collection of inferences and control rules. The control rules, which are together with the inferences also provided by MAYA, represent mathematical knowledge about how to proceed at choice points in the planning process, for example, which subgoals to tackle next or which inference to apply first. With the help of strategies the search space is organized and certain search paths are preferred and others are pruned. MULTI can flexibly interleave different strategies in a proof attempt.

Inferences can also encode some abstract-level proof ideas rather than low-level calculus rules and thus the proof plans delivered by MULTI are not always necessarily correct. However, abstract proof plans can be recursively expanded to logic level proofs within a verifiable calculus. If this expansion succeeds, a valid proof plan and a corresponding checkable low-level proof has been found. If the expansion fails, the proof plan remains invalidated. More information on multi-strategy proof planning can be found in [17].

Ω-Ants – The Agent based Suggestion Mechanism: The Ω-ANTS component supports interactive proof construction by generating a ranked list of bids of potentially applicable inferences in each proof situation. In this process, all inferences are uniformly viewed with respect to their arguments, that is, their premises, conclusions, and additional parameters. An inference rule is applicable if there is a sufficiently complete instantiation for its arguments. The task of Ω-ANTS is to determine the applicability of inference rules by incrementally computing instantiations for the arguments. These applicability checks are performed by separate processes, that is, software agents which compute and report bids. More details on Ω-ANTS can be found in [9, 10].

The task of the other components is to make ΩMEGA's functionalities available in different application scenarios. One application scenario is to support the writing of scientific publications in the WYSIWYG text-editor TEX$_{MACS}$ [15] with the ΩMEGA system running in the background. The idea is to provide formal analysis and verification means for the mathematical theories developed and authored in TEX$_{MACS}$. The integration of ΩMEGA and TEX$_{MACS}$ is thereby realized via the PLATΩ component.

PlatΩ – The Mediator: The PLATΩ system dynamically establishes and guarantees the consistency between the human-oriented representations in TEX$_{MACS}$ documents and the corresponding machine-oriented representations maintained in ΩMEGA.

Mediation between a text-editor and a proof assistance system requires the extraction of the formal content from the document of the text-editor. For this task the PLATΩ system employs semantic annotations in the TEX$_{MACS}$ document, which are pre-defined TEX$_{MACS}$ macros manually provided by the author. The annotations can be nested and they structure the text into dependent and independent parts containing assertions and (partial or complete) proofs encapsulating proof steps. From the concrete syntax in the TEX$_{MACS}$ document the PLATΩ system generates in a multiphase process an abstract syntax. This abstract syntax describes a formally structured theory with possibly partial formal proofs. The grammar for PLATΩ's formula parsers is thereby generated on the fly from the dynamic notation [4] information as provided by the author inside the TEX$_{MACS}$ document. In order to preserve consistency, the relationship between transformed knowledge objects is memorized. More information on PLATΩ is available in [26, 27].

ΩMEGA is furthermore exploited as a domain reasoner in the proof tutoring environment DIALOG [8]. In the tutorial dialogs as studied and realized in the DIALOG context, the user is given a proof exercise to be solved interactively with the system. The DIALOG system provides feedback to the student's solution attempts and aids the user in finding a solution, with the overall goal to convey specific concepts and techniques of a given mathematical domain. To support the generation of appropriate feedback, each proposed proof step needs to be analyzed by the system in the context of the partial proof developed so far. These tasks are addressed by the PROOF STEP VERIFIER and the GRANULARITY ANALYZER.

Proof Step Verifier: The PROOF STEP VERIFIER verifies underspecified, ambiguous, and incomplete proof steps as typically uttered by students in a tutorial dialog, thereby resolving underspecification and ambiguities whenever possible. For this task the PROOF STEP VERIFIER collaborates with the TASKLAYER and benefits from its ability to represent multiple alternative proof attempts for the same proof obligation simultaneously. This feature of the TASKLAYER is exploited to dynamically represent and maintain all proof states that are consistent with the ambiguous utterances of the student.

The PROOF STEP VERIFIER internally maintains the relationship between the proof step utterances obtained sequentially from the student and the corresponding proof parts in each alternative proof in the TASKLAYER. It exploits the dependency relations for instance for backtracking, when the student takes back a proof step or resumes her proof at an earlier stage. Once the student utters a new proof step, the PROOF STEP VERIFIER determines for each derived consistent proof state all possible successor proof states that are consistent with that utterance. This may result in even more alternative proofs if that utterance was ambiguous but it may also rule out certain proof alternatives that are no longer consistent with the student's utterance. More information on the PROOF STEP VERIFIER is available in [6].

Granularity Analyzer: The GRANULARITY ANALYZER evaluates the argumentative complexity of student proof steps in a tutorial context. The problem is that student proof steps may well be classified as correct by the PROOF STEP VERIFIER but nevertheless be unacceptable from a tutorial perspective since they are either to 'big' (to coarse-grained) or to 'small' (to detailed). The GRANULARITY ANALYZER thus categorizes the step size of proof steps performed by the student, in order to recognize if they are appropriate with respect to the given tutorial context. The result of the granularity analysis for an uttered proof step is a granularity judgment, which can take one out of three possible values: appropriate, too detailed, and too coarse-grained.

To compute its granularity judgment the GRANULARITY ANALYZER exploits the results of the proof step verification: It analyzes each student proof step with respect to (i) the total number of steps required to prove the step, (ii) the number of different concepts used in that proof, (iii) the number of concepts the student has used before as well as the number of concepts not used before, (iv) the number of new hypotheses introduced by the proof step, and (v) the number of introduced subgoals. For this, the GRANULARITY ANALYZER employs a machine learning approach in which individual preferences and domain dependencies in the granularity judgments can be learned.

2.2. Motivating Example

We illustrate the overall functioning of the ΩMEGA system and the roles of the components MAYA, TASKLAYER, MULTI, PLATΩ, and Ω-ANTS with an example. In this example student Eva wants to prove the commutativity of addition

$$\forall x, y.x + y = y + x \tag{2.1}$$

in the standard Peano axiomatization. Eva is typing this proof in the text-editor $\text{T}_{\text{E}}\text{X}_{\text{MACS}}$ (cf. Figure 2) while subsequently receiving assistance from the ΩMEGA system running in the background.

Phase 1. After having specified the theory and the conjecture in $\text{T}_{\text{E}}\text{X}_{\text{MACS}}$, the document is uploaded into ΩMEGA. That is, the whole document is passed to the PLATΩ component which extracts the formal content of the document including
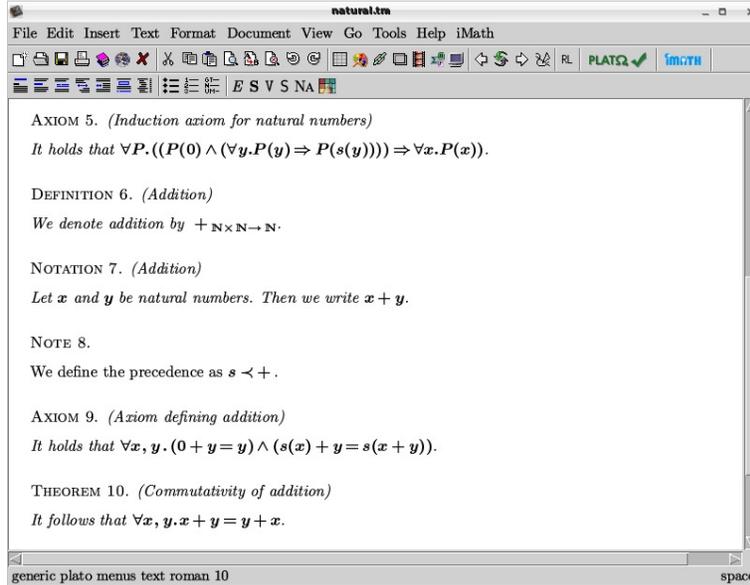
Figure 2: Formalization of the example scenario in the scientific T_EX_{MACS} editor.

notational information to parse formulas. From the document PLATΩ builds up the corresponding hierarchically structured theories including the open conjecture, that is, Theorem (2.1) (resp. "Theorem 10." in Figure 2). PLATΩ also computes and maintains the information which part of the document corresponds to which part of the structured theories. The latter is passed to the MAYA component, which generally maintains structured theories. MAYA passes the proof strategies, control rules, and symbol orderings to the proof planner MULTI. Furthermore, MAYA initializes a proof for the open conjecture in the TASKLAYER. Thereby, definitions, axioms, and lemmas are automatically compiled into derived inference rules and passed to the TASKLAYER as well. The TASKLAYER builds up a proof tree and forwards the inference rules to the proof planner MULTI and the suggestion component Ω-ANTS. MULTI then initializes a proof planning session for the open conjecture and the Ω-ANTS system creates suggestion agents for the obtained inferences.

Phase 2. Eva begins to prove the conjecture within T_EX_{MACS}. First, she requests a hint from ΩMEGA on how she could possibly proceed. This request is passed on via PLATΩ and the TASKLAYER to MULTI and Ω-ANTS. MULTI returns a list of available strategies (via TASKLAYER and PLATΩ) back to the T_EX_{MACS} editor. From this list Eva selects the strategy `InductThenSimplify`. This selection is reported (via PLATΩ and TASKLAYER) to MULTI which then executes the strategy and

applies an induction on x to the open conjecture in the TASKLAYER. Immediately afterwards, the strategy tries to simplify the resulting subgoals using a simplification tactic. Unfortunately, however, the strategy does not succeed in finishing the proof and terminates with the following two open subgoals: (2.1a) $z = z + 0$ and (2.1b) $s(x + z) = z + s(x)$. The partial proof for Theorem (2.1), maintained in the TASKLAYER, now contains the executed induction and simplification proof steps and these two new subgoals. These changes of the partial proof are compiled into patch descriptions for the proof representation and then passed to PLATΩ via MAYA. PLATΩ transforms the obtained tree-like subproof representation into a linear, text-style proof representation using pseudo-natural language. Moreover, PLATΩ renders the formulas using the memorized notational information. Finally, PLATΩ exploits the memorized relationship between structured theories and the document, to generate a patch description for the document which is then passed to TEX_MACS.

Phase 3. Now Eva is confronted with the partial proof and the two new subgoals. She quickly realizes that two lemmas are needed. Hence, she leaves the proof of the theorem as it is and adds the following two lemmas somewhere in the TEX_MACS document:

$$\forall x. x \quad = \quad x + 0 \tag{2.2}$$
$$\forall x, y. s(x + y) \quad = \quad y + s(x) \tag{2.3}$$

Similar as before, these lemmas are uploaded into ΩMEGA, except that now only patches are transferred. More concretely, a difference analysis is applied between the old and the new version of the document, and only the new lemmas are parsed and a patch description to add their formal counter-parts as open conjectures to the theory is send to MAYA. This, in turn, triggers the initialization of proofs in the TASKLAYER and the initialization of MULTI and Ω-ANTS, respectively.

Phase 4. Eva then tackles these lemmas one by one using the strategy `InductThen-Simplify` (again automatically suggested by MULTI) and succeeds in proving them fully automatically. The resulting proof descriptions are again transformed into proof patches that are included into the document via MAYA and PLATΩ. However, since the lemmas are now proved, their status, which is maintained in MAYA, is set to `proved`. This, in turn, causes MAYA to compile these lemmas and to pass them as new available knowledge for the ongoing proof of Theorem (2.1) to the TASKLAYER.

Phase 5. Eva continues the proof of (2.1) and the Ω-ANTS component now automatically suggests to apply the lemma (2.2) to the subgoal (2.1a) and the lemma (2.3) to the subgoal (2.1b). Eva selects these suggestions one by one, which then completes the proof. Note that the applications of the suggested lemmas are actually executed by the TASKLAYER, and, again, only the proof patch descriptions are transformed into the patches to the TEX_MACS document like before.

### 2.3. Ωmega's Realization as Component Network

The ΩMEGA system is composed of the state-full components introduced in Section 2.1. These components are linked to their neighboring components via bidirectional interfaces. Each component maintains and manipulates mathematical knowledge represented in some suitable data format. This includes knowledge received as input from other components and knowledge that is passed as output to other components. It also includes knowledge which is stored only internally and which is hidden from other components. Moreover, new knowledge is typically inferred from knowledge available within these components.

An important aspect in our approach is that the components maintain also connections and dependencies between the different kinds of knowledge they work with. The communication between components is realized by exchanging relevant knowledge in the form of documents. For this, each link in the network employs a link-specific, semi-structured *document format*. It is allowed that different links share the same document format. The bidirectional communication protocol between two network nodes, connected via a link of some document format is based on patches for documents of this format. Instead of assuming one general *patch description language* for all document formats, we allow each document format to provide its own one. From an abstract perspective the network components act as *document transformation functions*, one for each ordered pair of connected links.

If some change occurs in some component, this change needs to be propagated through the whole system. In our component network this means that document patches are propagated along the links. If a patch arrives in some component, it is transformed into patches for all links connected to this component, including the link from which it arrived.

For instance consider the following example fragment of ΩMEGA's component network:
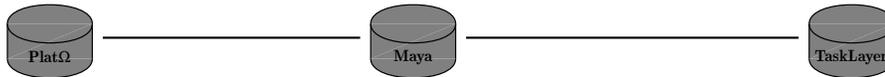
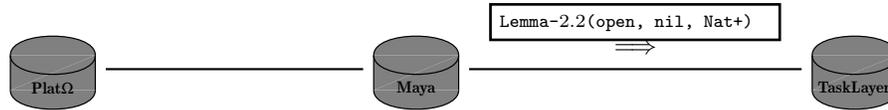After the student Eva has stated the new lemma (2.2) in our example session and started its proof, PLATΩ propagates the information to MAYA that there is a new lemma Lemma-2.2 with status open and whose associated proof is empty (nil). This information is encoded as Lemma-2.2(open, nil) and we depict the update as
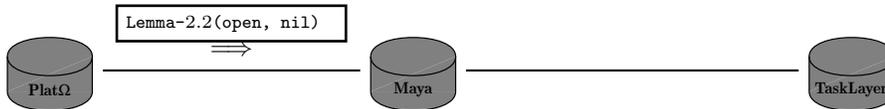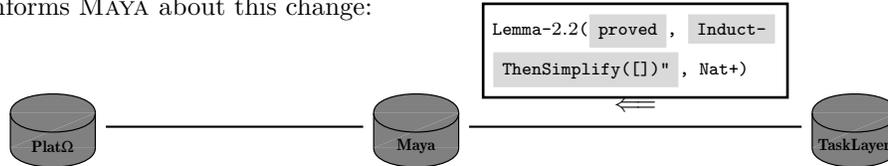

MAYA integrates the update and in turn notifies the TASKLAYER that there is a new lemma (2.2) which is open and which has an empty proof (nil). MAYA also adds the information that in order to prove the lemma the set of inferences

`Nat+` are available. They have been synthesized and compiled by MAYA from the axioms defining the naturals and addition. This update is depicted by:

```
                                    | Lemma-2.2(open, nil, Nat+) |
                                    |            ==>             |
   PlatΩ ─────────────── Maya ─────────────────────── TaskLayer
```

As soon as the suggested strategy `InductThenSimplify` has successfully proved lemma (2.2), the TASKLAYER updates the status information for lemma (2.2) and informs MAYA about this change:

```
                          | Lemma-2.2( proved , Induct- |
                          | ThenSimplify([])" , Nat+)   |
                          |           <==               |
   PlatΩ ──────────────── Maya ──────────────────── TaskLayer
```

Here we use the shading ⬛X to indicate information bits that have been modified. `"([])"` in `"InductThenSimplify([])"`indicates that the list of open subgoals after application of the strategy is now empty.
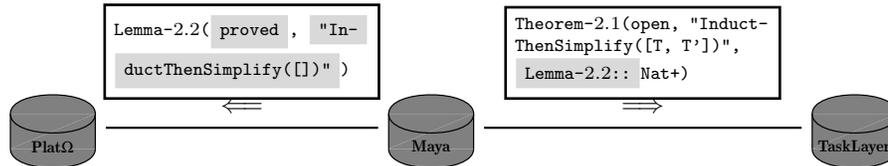
The update of the status information is further propagated by MAYA on the link to PLATΩ. The information about the usable inferences is thereby omitted:

$$\texttt{Lemma-2.2(proved, "InductThenSimplify([])")}$$

Since lemma (2.2) is now available for proving the main Theorem (2.1) MAYA furthermore updates the information on the link to the TASKLAYER by adding the inferences automatically synthesized from that lemma:

$$\texttt{Theorem-2.1(open, "InductThenSimplify([T, T'])", Lemma-2.2::Nat+)}$$

Here `T` denotes the open task for subgoal (2.1a) and `T'` the open task for subgoal (2.1b). These two updates occur simultaneously, which is depicted as:

```
  | Lemma-2.2( proved , "In-  |     | Theorem-2.1(open, "Induct- |
  | ductThenSimplify([])" )   |     | ThenSimplify([T, T'])",    |
  |                           |     | Lemma-2.2:: Nat+)          |
  |          <==              |     |           ==>              |
   PlatΩ ────────────── Maya ──────────────────────── TaskLayer
```

Without any additional control, the update procedure may run into conflicting situations, where one component receives updates simultaneously from different links. This phenomenon is similar to what can be observed in distributed databases, where different transaction processes run concurrently.

We solved the problem as follows: A subset of directly connected components is marked as the network's kernel components. The invariant is that the kernel components influence each other in a well-defined manner and mutual updates are propagated until no more updates are triggered by these components. For all other components of the network we impose that any change stemming from a

	Inferences	Tasks	Proof Knowledge (Strategies, Control Rules, Orderings, ...)	DG views	TL views
Maya-TaskLayer	✓				✓
PlatΩ-Maya				✓	✓
TaskLayer-Multi	✓				✓
Maya-Multi			✓		
TaskLayer-Ω-Ants	✓	✓			
Maya-Proof Step Verifier				✓	✓
TaskLayer-Gran. Analyzer					✓

Table 1: The fields marked by ✓ indicate the document (sub-)formats communicated and patched at the individual component interfaces.

component closer towards the kernel components never causes a change that must be propagated towards the network kernel. Assuming such a network structure and invariants, a change initially triggered at some arbitrary component is digested in two phases as follows: In a first phase, the updates are only propagated towards the kernel components. Any change triggered by intermediate components are also only relayed towards the kernel components and the updates to any other component are put on hold. Once the kernel components are reached, their reciprocal updates are performed only among these components and until the system stabilizes. Only then the updates are accumulated for the non-kernel components and propagated outwards through the network in a second phase.

Initially, the system kernel consists of the Maya system and the TaskLayer. However, to allow for a more efficient propagation of the knowledge during certain phases, the kernel components can also change. As we will see later, the kernel components can also consist of the TaskLayer and the Multi component.

3. Interfaces and Interplay of Components

We now present in more detail the interfaces between the different components of Ωmega as depicted in Figure 1. For each interface, we describe the kind of knowledge exchanged over that interface as well as the patch operations used by the involved components to update the knowledge. The knowledge exchanged over the interfaces comes in specific document formats. Typically they are abstractions of the internal representations of the individual components. We first introduce the different document formats in Section 3.1 before describing the individual interfaces and patch operations in Section 3.2. Using our example scenario from Section 2.2 we illustrate in Section 3.3 how the knowledge is propagated along the individual interfaces in each phase of the example.

3.1. The Document Formats used at the Interfaces

Different document formats are used at the individual interfaces and each document format may be a combination of other document formats. Table 1 gives an overview which document formats are used at which interface.

The *inferences* are the inferences of the TASKLAYER and the *tasks* are the open subgoals of the proof trees maintained in the TASKLAYER. The *proof knowledge* entails the strategy descriptions, control rules and symbol orderings used by the proof planner. All these concepts have already been introduced in Section 2.1. We now define the notions of TASKLAYER views (*TL views*) and Development Graph views (*DG views*).

TL views. The TASKLAYER supports the representation of alternative proof steps for *both* the (horizontal) reduction of a goal as well as for the (vertical) expansion of a complex proof step to higher granularity. For the user or a component such as MAYA not all available information on these alternatives is typically needed or desired. For this purpose the information being communicated is drastically reduced, by means of a *TL view*, which selects only one horizontal layer of a (partial or complete) proof, that is, a proof at a particular level of granularity. This horizontal layer, however, contains all (horizontal) OR-alternatives of the proof. Thus, TL views are (partial) proof trees where each subtree is annotated by a task and a list of alternative justifications for that task.[1] A justification is a list[2] of proof trees for subtasks together with a description how the task can be justified from the subtasks. Furthermore, a justification provides status information, such as whether that justification has been verified and if more details about its verification are available.

DG views. Remember that the internal representation of the MAYA system are development graphs. They contain the hierarchically structured axiomatic theories including the conjectures as well as global proof obligations. MAYA also maintains dependency information, such as how global proof obligations are decomposed into local proof obligations and which assertion has been used in which proof. Finally, MAYA maintains proof knowledge in various forms: (a) predefined inferences, such as calls to external systems, (b) predefined, domain-specific or problem-specific strategies and control rules that are used by MULTI (c) derived information such as the orderings among functions extracted from the definitions of these symbols. Only a subset of all this information is interesting to communicate with other components. Therefore we define *Development Graph views* (*DG views*) as a subset of MAYA's internal representation that consists of (i) the structured theories, (ii) their signature declarations, axioms, and conjectures, (iii) the global proof obligations without decomposition information.

---

[1] If the list of justifications is empty, then the task is an open goal.
[2] This list can be empty, for instance in case of the axiom rule.

3.2. The Interfaces

We now describe the patch operations used at the individual interfaces.

The Maya–TaskLayer Interface. Maya maintains the hierarchical theories and the TaskLayer their proofs. The information exchanged at the interface between these components consists of all conjectures in all theories along with their currently available inferences and their TL view. The patch operations for the inferences are addition and deletion. The patch operations for TL views are addition and deletion of proof steps, replacement of fragments of proof trees by other proof tree fragments and modification of the status information of proof steps. The replacement of tree fragments is not restricted to proper subtrees and can be applied to arbitrary fragments. This supports the selective and flexible change of proof step granularities by a simple patch replacing a number of abstract proof steps by their subproofs with a higher granularity or vice versa. Changing the proof step granularity is a feature supported by the TaskLayer like the simultaneous representation of (OR-)alternative sub-proofs.

The PlatΩ–Maya Interface. The information exchanged on that interface are hierarchically structured theories. Each theory in Maya extends the imported underlying theories by signature declarations for new sorts and constants, axioms and conjectures together with their TL views. The non-TL view parts of the theory representation are the DG views and we denote the combined structured theory representation as *DG+TL views*. The patch operations for DG+TL views include: the addition or deletion of whole theories or theory elements such as sort declarations, constant declarations, axioms and conjectures. It further includes the patch operation on TL views as already described for the interface between Maya and the TaskLayer. In contrast to the patch operations for replacement of fragments of TL views, we currently do not support the modification of other theory elements, such as changing an axiom by changing the formula. However, such an extension is a hot topic for future work and will be discussed in Section 4.

The TaskLayer–Multi Interface. For each conjecture the TaskLayer maintains simultaneously a proof at different (vertical) granularity levels and with OR-alternatives (horizontal). From each such complex proof object, Multi receives from the TaskLayer a proof at one level of granularity, but with all OR-alternatives, that is a TL view. The decision on which alternative proof the proof search should proceed is part of Multi's strategical proof search mechanism. For each TL view, Multi also obtains all available inferences from the TaskLayer. Hence, the interface representation is a list of triples that consist of conjectures along with their corresponding TL view and available inferences. The patch operations are the addition or deletion of entire triples, the addition or deletion of inferences and the patch operations for TL views.

The Maya–Multi Interface. MAYA maintains for each theory not only the available assertions, but also domain specific inferences (such as calls to external systems), strategy descriptions, control rules or symbol orderings. For each of its conjectures MAYA makes this information available on the interface to MULTI. Aside from adding and deleting whole entries, the patch operations are the deletion or addition of elements for the different fields of an information record, like, for instance, the addition of a strategy description or the ranking of two symbols.

The TaskLayer–Ω-Ants Interface. The Ω-ANTS system obtains for each open task in each proof maintained by the TASKLAYER the list of inferences available for that task. The patch operations consist of adding and deleting open tasks as well as inferences. Consider as an example an inference that has been applied by the TASKLAYER on some open task and which resulted in two new tasks: the Ω-ANTS system is informed to delete the old task, to add the two new tasks and to search now for suggestions how to apply inferences on the latter.

The Maya–Proof Step Verifier Interface. This interface has the same representation and patch operations as the interface between PLATΩ and MAYA. However, only a subset of the available patch operations are currently used. At present, the PROOF STEP VERIFIER uploads the DG view that sets up the context of a proof tutor exercise and then only adds or deletes individual proof steps. In the other direction, MAYA simply changes the status information of proof steps (checked vs. unchecked).

The TaskLayer–Granularity Analyzer Interface. This interface uses TL views as the interface representation and the corresponding patch operations for TL views. The interface is currently unidirectional, since the GRANULARITY ANALYZER never changes the TL view so that changes are only propagated from the TASKLAYER to the GRANULARITY ANALYZER.

3.3. Illustration of Information Flow

We now illustrate the functioning of the whole system by describing for each phase of the example scenario (Section 2.2, p. 7) which patches are propagated via the interfaces through the network.

Phase 1. After Eva has edited the formalizations in $\TeX_{MACS}$ and stated the main theorem, she uploads the document into PLATΩ. PLATΩ parses the document and creates a corresponding DG+TL view for it, that consists of a single theory with signature declarations and axioms and one conjecture with an empty proof. It patches the representation on its interface with MAYA, which is depicted by $\overset{1.1}{\Longrightarrow}$ in Figure 3.

After updating its internal representation, MAYA patches the document at the interface with the TASKLAYER by adding a new conjecture (Theorem (1.2)), an empty proof for it and the list of inferences that are available to prove the
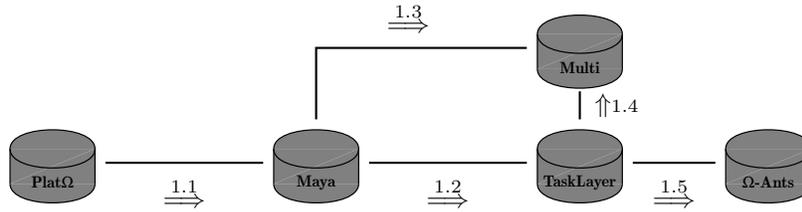
Figure 3: Flow of document patches in Phase 1

conjecture. The inferences have been synthesized from the available axioms. Furthermore, Maya patches its representation at the interface with Multi by adding all available strategies, control rules and symbol orderings for the new conjecture.

The TaskLayer sets up a new active proof tree, and forwards the single open tasks and inferences to the Ω-Ants system (1.5) and the corresponding TL view and the related inferences to Multi (1.4). The Ω-Ants system initializes a new agent society for the new open task and Multi starts a new proof planning session based on the TL view and the received inferences. Multi also includes the additional proof knowledge available for that conjecture obtained via its interface with Maya (1.3).

Phase 2. Eva invokes the `InductThenSimplify` strategy provided by Multi, which initially suggests to apply the inference that stems from the induction axiom. The application request is passed over to the TaskLayer. At the same time, the kernel components are switched temporarily to consist of the TaskLayer and Multi. This allows Multi to apply multiple inferences before the patches are propagated to the other connected components (Maya and Ω-Ants), which avoids unnecessary updates. The TaskLayer first applies the inference requested by Multi and patches the TL view towards Multi (patch 2.1 in Figure 4).
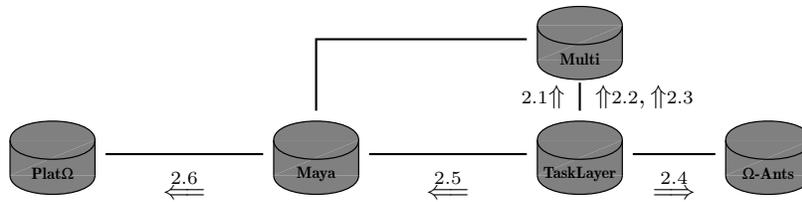

Figure 4: Flow of document patches in Phase 2

After that, Multi requests the application of $X + 0 = X$ and $X + S(Y) = S(X+Y)$ on either subgoal which gives rise to the patches 2.2 and 2.3. Finally, the

`InductThenSimplify` strategy terminates. The kernel components are set back to consist of MAYA and the TASKLAYER, and the accumulated patches are propagated further: the old open task for Ω-ANTS is removed and the two new open tasks are instead propagated to Ω-ANTS (2.4). Furthermore, the patch of the TL view is applied on the interface to MAYA (2.5) and further propagated to PLATΩ (2.6). PLATΩ then integrates a description of the three new proof steps in pseudo-natural language into the T$_{E}$X$_{MACS}$ document. This completes the processing of Phase 2.

Phase 3. In this phase Eva adds the lemmas (2.2) and (2.3) to the document. The updates are propagated analogously to the updates in Phase 1, except that the patch consists now only of the two additional conjectures with empty proofs. These are inserted into the DG+TL view by PLATΩ and propagated to MAYA (3.1 in Figure 5).
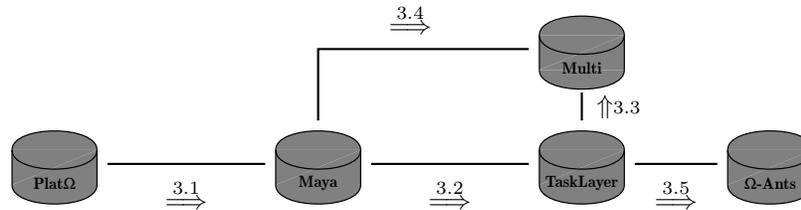

Figure 5: Flow of document patches in Phase 3

MAYA forwards the empty TL views to the TASKLAYER together with the available inferences (3.2) and makes the strategies, control rules and ordering information for these conjectures available to MULTI (3.4). The TASKLAYER initializes two new proof trees and forwards the respective patches to MULTI (3.3), which creates two new proof planning sessions. The TASKLAYER also forwards these patches to Ω-ANTS (3.5), which initializes the suggestion agents for the two new open tasks.

Phase 4. Now Eva applies the `InductThenSimplify` strategy to both lemmas. Similar to Phase 2, this triggers the repeated propagation of TL view updates from the TASKLAYER to MULTI. We have summarized this by the single ⇑ 4.1 in Figure 6. Different to Phase 2 the strategies now succeed in proving the two lemmas. The information that these two proofs are complete is propagated to Ω-ANTS (4.2), which then removes the open tasks and the suggestion agents, and to MAYA (4.3). MAYA forwards the TL view patches for both lemmas to PLATΩ (4.4). Furthermore, since the lemmas are now proved, the internal management of MAYA makes these lemmas subsequently available for the interrupted proof of the main theorem. More concretely, it synthesizes the inferences for these two lemmas and adds them to the available inferences for the main theorem by patching the
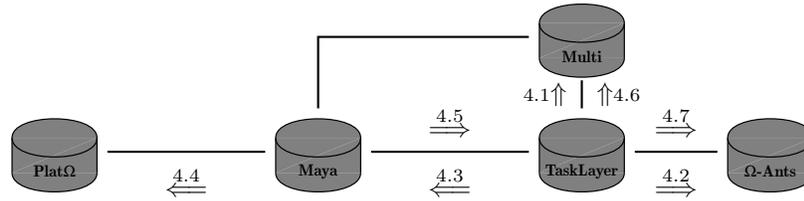
Figure 6: Flow of document patches in Phase 4

document at the interface to the TASKLAYER (4.5). The TASKLAYER propagates these new inferences to MULTI (4.6) and for both open tasks of the main proof to Ω-ANTS (4.7). Ω-ANTS creates and includes the suggestion agents for these inferences and is hence able to suggest the application of the lemmas in the next phase.

Phase 5. Eva selects the two inferences suggested by Ω-ANTS for the two subgoals and requests the TASKLAYER to apply them. The TASKLAYER applies both which completes the proof of the main theorem. It also informs the Ω-ANTS system to delete both open tasks (5.1, Figure 7) and patches the TL view for MULTI
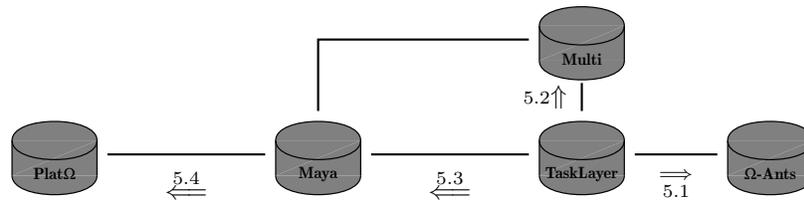

Figure 7: Flow of document patches in Phase 5

(5.2) and MAYA (5.3) accordingly. MAYA in turn forwards the TL view patches to PLATΩ, which integrates a pseudo-natural language description for the two new proof steps into the TeX$_{\text{MACS}}$ document. Since the theorem is now proved, MAYA checks whether it should make it subsequently available for other proofs, like the proved lemmas in Phase 4. However, since the proofs maintained in the TASK-LAYER for these two lemmas contributed to establishing the main theorem, the theorem is not propagated to them. This completes the description of how updates of interface documents are propagated in each phase of our running example.

4. Discussion

Why is the propagation of mathematical knowledge by non-monotonic patches a good choice? To discuss this we consider the use case of interactive mathematical authoring with $\text{T}_{\text{E}}\text{X}_{\text{MACS}}$. In this scenario a mathematician develops a mathematical document with definitions, axioms, theorems and proofs in the text-editor. Efficient propagation of mathematical knowledge is essential here since the document needs to be continuously synchronized with its formal counterpart in the proof assistance system. If we always performed a global transformation, we would rewrite the whole document in the text-editor. Consequently we would lose large parts of the natural language text written by the user. On the other hand we would lose the verification previously performed by the proof assistance system. For example, any already verified proof or any computation result from external systems would be lost. Clearly, propagating the changes between both sides preserves most of the work done on either side.

Can we do better than just computing and propagating syntactic changes? Currently, the document in the text-editor is structured which enables the mediator PLATΩ to extract a tree-structured representation. *Structured* means here, for example, that proof steps are classified into assumptions, facts, subgoal introductions and other steps. Additionally, the syntactic formulas are converted into structural formulas making variables and operators explicit. Any syntactic change of the document is reflected in a change of the tree-structure and would trigger changes inside the proof assistance system–even if the change was only a reorganization of the document and semantically irrelevant.

How can we use the semantics to optimize the differencing mechanism? We integrated the differencing mechanism XMLDIFF [21] that can be parameterized by equivalence classes for structural elements. This way we are able to efficiently deal with *document rearrangement*: A proof has in general to be re-verified when proof steps are permuted but not if the order of subgoals or the order of their subproofs changes. The latter is in fact only syntax sugaring and irrelevant to the formal verification. Although the document syntactically changes in that case, such a modification is filtered out by specifying the equivalence class for subgoal introductions in a way that their content, that is the subgoals und subproofs, are orderless. Furthermore, for the formal verification all definitions, axioms and theorems are visible inside the whole theory. Hence, a rearrangement has no effect and does not need to be propagated from the text-editor to the proof assistance system.

Can the semantics-based differencing be used for further improvements of the authoring process? In [19] we describe a possible extension of the semantics-based approach to ontology-driven management of change aiming at the support of *document refactoring*. If an author for example modifies the formula $\forall A, B.A = B \Leftrightarrow A \subset B \wedge B \subset A$ to the formula $\forall \boxed{C}, B.A = B \Leftrightarrow A \subset B \wedge B \subset A$, the modified variable (indicated by $\boxed{C}$) is identified and instead of propagating this modification to the proof assistance system we preview the renaming effects by displaying the

formula $\forall \boxed{C}, B.\boxed{C} = B \Leftrightarrow \boxed{C} \subset B \wedge B \subset \boxed{C}$. By improving consistency on the document level the semantic differencing should also act like a firewall blocking erroneous input to prevent unnecessary verification of inconsistent input. When the author modifies the initial formula in our example to $\forall \boxed{C}, B.\boxed{D} = B \Leftrightarrow A \subset B \wedge B \subset A$, the automatic adaptation fails in the former occurrence of the variable A that has been renamed to D. Therefore the author will be asked whether or not this conflict is intended. Moreover, by identifying dependent modifications we are able to return a combined meta change information. Considering the example with the old formula $\forall A, B.A = B \Leftrightarrow A \subset B \wedge B \subset A$ and the new formula $\forall \boxed{C}, B.\boxed{C} = B \Leftrightarrow \boxed{C} \subset B \wedge B \subset \boxed{C}$, we propagate the α-conversion of the variable A to C as *replacement* $\{A \mapsto C\}$ instead of propagating the renaming of each single occurrence of the variable A in that formula. The implementation of this extension of the management of change mechanism is on the way. The main obstacle for the support of evolutionary proof authoring is that changes may invalidate proofs, which then need to be rebuilt using an inhibitive amount of resources. Therefore we will investigate whether a set of edit operations in the text-editor can be classified into preconceived transformations [22] that are operating on the state of the formal development, motivated by the question: *How would one patch the proofs on paper given a consistent transformation?*

Although mathematical knowledge management clearly benefits in our scenario from the presented techniques, there is room for many improvements. For example, the differencing mechanism could be extended to detect the replacement of equivalent subterms. Adding a property to a definition could automatically result in a new case popping up in an existing proof rather than pruning and reworking that proof.

What is required to efficiently deal with changes? If the components support only the change operators *addition* and *deletion* the system only supports a limited management of change. However, efficient *document refactoring* requires the system-wide support of the change operator *update*. Otherwise, proofs are always pruned when definitions used are slightly modified, even if a simple proof repair is possible.

Is it always reasonable to propagate arbitrary small changes? The more elaborate the semantics-based differencing mechanism gets, the more expensive it gets. We need to develop a framework for estimating these costs in order to decide on which granularity level changes have to be propagated. The level of granularity also depends on the least common level supported by all system components. For example, it does not make sense to compute subterm differences, like variable renaming, for formulas in the text-editor when the proof assistance system only supports the replacement of whole formulas.

5. Related Work

The design of the new Ωmega system as a component network shares many aspects of *service oriented architectures* (SOA) [16]. In fact, we follow and apply many prominent SOA principles: A main objective in our work has been to modularize the overall mathematical assistance system into reusable individual autonomous components of appropriate size. Our components single out and encapsulate well defined subtasks of the overall mathematical assistance system and they hide and abstract the component-internal reasoning and knowledge maintenance/management processes from the outer world. The components are interoperable and depend on the specific application context (interactive theorem proving inside TeX_MACS or proof tutoring in DIALOG). They are composed in a suitable way so that they can optimally serve the specific needs. Different to SOA we do not employ service contracts but develop own mathematical document formats to serve our specific needs. Moreover, our mathematical documents so far are not compliant to industrial standards.

The way proof updates are handled in our approach differs to the best of our knowledge significantly from the way they are handled in other proof assistants, for example, ISABELLE [20], CoQ [25], or HOL [14]. Two prominent user interfaces for such proof assistants are CTCoQ [11] and PROOFGENERAL[2]. In these systems the user develops a proof of a particular theorem of a given theory in form of a proof script by textually typing commands in an ASCII text editor. Feedback in form of information about the open goals and other messages are sent to separate buffers of the text editor. Proof scripts are composed of commands and they can be stepwise executed by moving the *execution point* of the proof script. The already executed parts of a proof script are thereby locked to avoid accidental editing. The current execution point is visible in the document. An undo step moves the execution point one step backwards.

In our approach a proof step or an undo operation corresponds to a document update which allows for more general updates as is possible in the work cited above:

Locality of Proof Script Updates. In CTCoQ and PROOFGENERAL an update can only be performed at the current execution point, while in our approach the user can directly edit arbitrary parts of the proof script, in which case a corresponding patch description is sent to the proof assistance system.

Parallel Editing of Proof scripts. Our approach allows the user to edit the proofs of several theorems simultaneously, which is not possible if updates are restricted to one reference point only. In the latter case there is obviously no need for an advanced update mechanism for proved theorems, since the assertions that are available at a specific reference point are always those which are available in the theory plus those proved in the document before the reference point. In our approach we have to deal with more complex situations and rely on the elaborate truth maintenance capabilities of the proof assistance system.

A more sophisticated theory update mechanism that comes close to our approach is implemented in MATITA [1], which is an interactive theorem prover, that organizes the mathematical knowledge in a searchable knowledge base. To ensure consistency of the library, MATITA employs two mechanisms called *invalidation* and *regeneration*. If a mathematical concept is changed, the concept itself and all concepts depending on it are invalidated and need to be regenerated to verify whether they are still valid. To regenerate an invalidated part of the library, MATITA re-executes the scripts that produced the invalidated concepts.

In our approach we do not invalidate the complete proofs, but only those proof steps that depended on a changed part: for these parts we would also have to re-execute the scripts. However, we refrain to do so, because re-execution of scripts is time consuming and in our interactive settings the response time of the system is crucial. Therefore our objective has been to extend the update mechanisms to propagate modifications and develop mechanisms to repair proofs locally depending on the kinds of modification.

6. Conclusion

We have presented the component network based design and implementation of the new ΩMEGA system, and we have described the mathematical knowledge management and transformation techniques required to organize the overall functioning of the system in different application contexts. The challenge was that throughout the system various kinds of given and derived knowledge occur in different formats and with different dependencies. These pieces of knowledge and their dependencies need to be maintained and, if changes occur in some component, they need to be effectively propagated.

The approach described in this article enables a rather flexible and independent addition of new system functionalities while assuring a maintainable system code by clear separation of concerns and decomposition of the system into individual components with its own internal representations. The architecture provides an adequate basis for truth maintenance techniques that accommodates the non-monotonic evolution of mathematical knowledge by enforcing the propagation of updates as the basic communication means between components. To our knowledge, basing the processing style of a whole mathematical assistance system on a non-monotonic update style rather than an incremental input processing style has not been addressed as systematically as in our approach.

Future work includes further investigation of the component network architecture and, in particular, the replacement of the current pragmatic change propagation mechanism by a general mechanism exploiting structural properties of the network. We also want to study how updates can be more efficiently propagated through the network and explore further techniques to merge updates obtained from different components. Moreover, we plan to integrate a service-passing mechanism where functionalities of remote components can be requested via the

network. On the application side, we plan to investigate domain-specific change protocols based on typical transformations of the mathematical knowledge (see for example [22]) towards supporting *mathematical knowledge refactoring*.

References

[1] A. Asperti, C. Sacerdoti Coen, E. Tassi, and S. Zacchiroli. User interaction with the MATITA proof assistant. *J. Autom. Reasoning*, 39(2):109–139, 2007.

[2] D. Aspinall. PROOFGENERAL: A generic tool for proof development. In S. Graf and M. I. Schwartzbach, editors, *TACAS*, volume 1785 of *Lecture Notes in Computer Science*, pages 38–42. Springer, 2000.

[3] S. Autexier, C. Benzmüller, A. Fiedler, H. Horacek, and B. Vo. Assertion-level proof representation with under-specification. *Electronic Notes in Theoretical Computer Science*, 93:5–23, 2004.

[4] S. Autexier, A. Fiedler, T. Neumann, and M. Wagner. Supporting user-defined notations when integrating scientific text-editors with proof assistance systems. In M. Kauers, M. Kerber, R. Miner, and W. Windsteiger, editors, *Towards Mechanized Mathematical Assistants*, LNAI. Springer, June 2007.

[5] 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.

[6] C. Benzmüller, D. Dietrich, M. Schiller, and S. Autexier. Deep inference for automated proof tutoring? In *KI 2007: Advances in Artificial Intelligence, 30th Annual German Conference on AI*, pages 435–439. Springer, 2007.

[7] C. Benzmüller, A. Fiedler, A. Meier, M. Pollet, and J. Siekmann. Omega. In *The seventeen provers of the world*, number 3600 in LNAI, pages 127–141, 2006.

[8] C. Benzmüller, H. Horacek, I. Kruijff-Korbayova, M. Pinkal, J. Siekmann, and M. Wolska. Natural language dialog with a tutor system for mathematical proofs. In *Cognitive Systems*, volume 4429 of *LNAI*. Springer, 2007.

[9] C. Benzmüller and V. Sorge. A blackboard architecture for guiding interactive proofs. In F. Giunchiglia, editor, *Proceedings of 8th International Conference on Artificial Intelligence: Methodology, Systems, Applications (AIMSA'98)*, number 1480 in LNAI. Springer, 1998.

[10] C. Benzmüller and V. Sorge. Ω-ANTS – An open approach at combining Interactive and Automated Theorem Proving. In M. Kerber and M. Kohlhase, editors, *8th Symposium on the Integration of Symbolic Computation and Mechanized Reasoning (Calculemus-2000)*. AK Peters, 2000.

[11] Y. Bertot. The CtCoq system: Design and architecture. *Formal Aspects of Computing*, 11(3):225–243, 1999.

[12] D. Dietrich. The TASKLAYER of the ΩMEGA system. Diploma thesis, FR 6.2 Informatik, Universität des Saarlandes, Saarbrücken, Germany, 2006.

[13] G. Gentzen. *The Collected Papers of Gerhard Gentzen (1934-1938)*. Edited by Szabo, M. E., North Holland, Amsterdam, 1969.

[14] M. J. C. Gordon and T. F. Melham, editors. *Introduction to HOL: a theorem proving environment for higher order logic*. Cambridge University Press, New York, NY, USA, 1993.

[15] J. van der Hoeven. GNU T$_E$X$_{MACS}$: A free, structured, WYSIWYG and technical text-editor. In D. Flipo, editor, *Le document au XXI-ième siècle*, volume 39-40, pages 39–50, Metz, 2001. Actes du congrès GUTenberg.

[16] F. Leymann and D. Karastoyanova (eds.). Service oriented architecture - overview of technologies and standards. *it - Information Technology (Special Topic Issue)*, 50(2), 2008.

[17] A. Meier, E. Melis, and J. Siekmann. Proof planning with multiple strategies. Artificial Intelligence, 2000.

[18] 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.

[19] N. Müller and M. Wagner. Towards improving interactive mathematical authoring by ontology-driven management of change. In *Proceedings of the Conference Wissens- und Erfahrungsmanagement (LWA'07)*, Halle/Saale, Germany, September 2007.

[20] T. Nipkow, L. C. Paulson, and M. Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS*. Springer, 2002.

[21] S. Radzevich. Semantic-based diff, patch and merge for xml-documents. Master thesis, Saarland University, Saarbrücken, Germany, April 2006.

[22] A. Schairer. *Transformations of Specifications and Proofs to Support an Evolutionary Formal Software Development*. Shaker Verlag, Aachen, 2006.

[23] J. Siekmann and S. Autexier. Computer supported formal work: Towards a digital mathematical assistant. In R. Matuszewski and A. Zalewska, editors, *From insight to proof - Jubilee Book for Andrzej Trybulec*, volume 10(23) of *Studies in Logic, Grammar and Rhetoric*, pages 231–248. University of Bialystok, July 2007.

[24] J. Siekmann, C. Benzmüller, and S. Autexier. Computer supported mathematics with ΩMEGA. *Journal of Applied Logic, special issue on Mathematics Assistance Systems*, 4(4), 2006.

[25] Coq Development Team. *The Coq Proof Assistant Reference Manual*. INRIA. see `http://coq.inria.fr/doc/main.html`.

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

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

[28] 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.

Acknowledgment

Serge Autexier
e-mail: `serge.autexier@dfki.de`
FR 6.2 Informatik, Saarland University & German Research Centre for Artificial Intelligence (DFKI GmbH), Saarbrücken, Germany

Christoph Benzmüller
e-mail: `chris@ags.uni-sb.de`
FR 6.2 Informatik, Saarland University, Saarbrücken, Germany

Dominik Dietrich
e-mail: `dietrich@ags.uni-sb.de`
FR 6.2 Informatik, Saarland University, Saarbrücken, Germany

Marc Wagner
e-mail: `wagner@ags.uni-sb.de`
FR 6.2 Informatik, Saarland University, Saarbrücken, Germany & Studienstiftung des Deutschen Volkes