Crafting a Proof Assistant

Andrea Asperti, Claudio Sacerdoti Coen, Enrico Tassi, and Stefano Zacchiroli

Department of Computer Science, University of Bologna
Mura Anteo Zamboni, 7 – 40127 Bologna, ITALY
{asperti,sacerdot,tassi,zacchiro}@cs.unibo.it

Abstract. Proof assistants are complex applications whose development has never been properly systematized or documented. This work is a contribution in this direction, based on our experience with the development of Matita: a new interactive theorem prover based—as Coq—on the Calculus of Inductive Constructions (CIC). In particular, we analyze its architecture focusing on the dependencies of its components, how they implement the main functionalities, and their degree of reusability.
The work is a first attempt to provide a ground for a more direct comparison between different systems and to highlight the common functionalities, not only in view of reusability but also to encourage a more systematic comparison of different softwares and architectural solutions.

1 Introduction

In contrast with automatic theorem provers, whose internal architecture is in many cases well documented (see e.g. the detailed description of Vampire in [16]), it is extremely difficult to find good system descriptions for their interactive counterpart. Traditionally, the only component of the latter systems that is suitably documented is the *kernel*, namely the part that is responsible for checking the correctness of proofs. Considering that:

1. most systems (claim to) satisfy the so called "De Bruijn criterion", that is the principle that the correctness of the whole application should depend on the correctness of a sufficiently small (and thus reliable) kernel *and*
2. interactive proving *looks like* a less ambitious task than fully automatic proving (eventually, this is the feeling of an external observer)

one could easily wonder where the complexity of interactive provers comes from.[1] Both points above are intentionally provocative. They are meant to emphasize that: (1) the kernel is possibly the most crucial, but surely not the most important component of interactive provers and (2) formal checking is just one of the activities of interactive provers, and probably not the most relevant one.

Of course, interactivity should be understood as a powerful integration rather than as a poor surrogate of automation: the user is supposed to interact when the system fails alone. Interaction, of course, raises a number of additional themes that are not present (or not so crucial) in automatic proving:

[1] e.g.: Coq is about 166,000 lines of code, to be compared with 50,000 lines of Otter

– library management (comprising both per-proof history and management of incomplete proofs);
– development of a strong linguistic support to enhance the human-machine communication of mathematical knowledge;
– development of user interfaces and interaction paradigms particularly suited for this kind of applications.

While the latter point has received a renewed attention in recent years, as testified by several workshops on the topic, little or no literature is available on the two former topics, hindering a real progress in the field.

In order to encourage a more systematic comparison of different software and architectural solutions we must first proceed to a more precise individuation of issues, functionalities, and software components. This work is meant to be a contribution in this direction. In particular we give in Section 2 a data-oriented high-level description of our interactive theorem prover denominated "Matita".[2] We also try to identify the logic independent components to understand the degree of coupling between the system architecture and its logical framework. In Section 3 we provide an alternative presentation of the architecture, based on the offered functionalities. Section 4 is an estimation of the complexity of the components and of the amount of work required to implement them.

Although our architectural description comprises components that (at present) are specific to our system (such as the large use of metadata for library indexing) we believe that the overall design fits most of the existent interactive provers and could be used as a ground for a deeper software comparison of these tools.

2 Data-Driven Architectural Analysis

Formulae and proofs are the main data handled by an interactive theorem prover. Both have several possible representations according to the actions performed on them. Each representation is associated with a data type, and the components that constitute an interactive theorem prover can be classified according to the representations they act on. In this section we analyze the architecture of Matita according to this classification.

We also make the effort of identifying the components that are logic independent or that can be made such abstracting over the data types used for formulae and proofs. This study allows to quantify the efforts required in changing the underlying logic of Matita for the sake of experimenting with new logic foundations while preserving the technological achievements.

The proof and formulae representations used in Matita as well as its general architecture have been influenced by some design commitments: (1) Matita is heavily based on the Curry-Howard isomorphism. Execution of procedural and declarative scripts produce proof terms (λ-terms) that are kept for later processing. Even incomplete proofs are represented as λ-terms with typed linear

[2] "matita" means "pencil" in Italian: a simple, well known, and widespread authoring tool among mathematicians

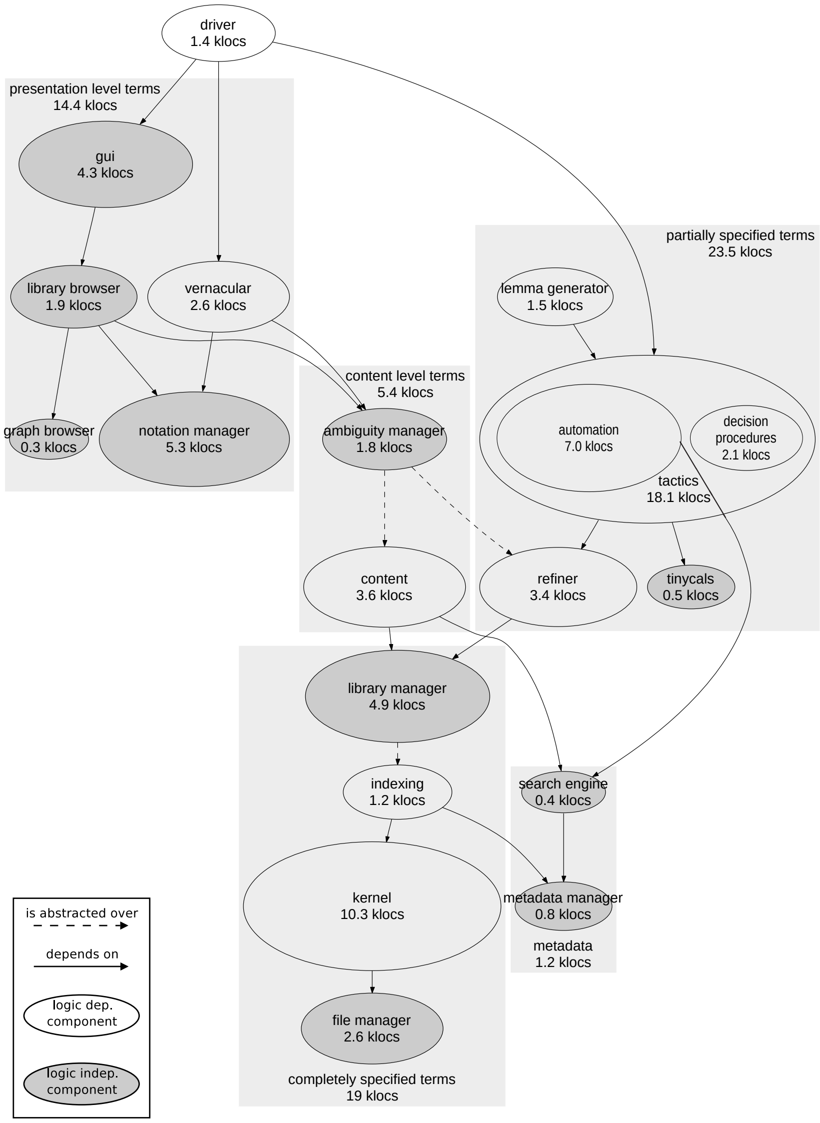

Fig. 1. Matita components with thousands of lines of code (*klocs*)

placeholders for missing subproofs. (2) The whole library, made of definitions and proof objects only, is searchable and browsable at any time. During browsing proof objects are explained in pseudo-natural language. (3) Proof authoring is performed editing either procedural or declarative scripts. Formulae are typed using ambiguous mathematical notation. Overloading is not syntactically constrained nor avoided using polymorphism.

According to the above commitments, in Matita we identified 5 term representations: presentation terms (concrete syntax), content terms (abstract syntax trees with overloaded notation), partially specified terms (λ-terms with placeholders), completely specified terms (well typed λ-terms), metadata (approximations of λ-terms).

Figure 1 shows the components of Matita organized according to the term representation they act on. For each component we show the functional dependencies on other components and the number of lines of source code. Dark gray components are either logic independent or can be made such by abstraction. Dashed arrows denote abstractions over logic dependent components. A normal arrow from a logic dependent component to a dark gray one is meant to be a dependency over the component, once it has been instantiated to the logic of the system.

We describe now each term representation together with the components of Matita acting on them.

Completely Specified Terms Formalizing mathematics is a complex and onerous task and it is extremely important to develop large libraries of "trusted" information to rely on. At this level, the information must be completely specified in a given logical framework in order to allow formal checking. In Matita proof objects are terms of the Calculus of Inductive Constructions (CIC); terms represent both formulae and proofs. The proof-checker, implemented in the *kernel* component, is a CIC type-checker. Proof objects are saved in an XML format that is shared with the Coq Proof Assistant so that independent verification is possible.

Mathematical concepts (definitions and proof objects) are stored in a distributed library managed by the *file manager*, which acts as an abstraction layer over the concept physical locations.

Concepts stored in the library are indexed for retrieval using metadata. We conceived a logic independent metadata-set that can accommodate most logical frameworks. The logic dependent *indexing* component extracts metadata from mathematical concepts. The logic independent searching tools are described in the next section.

Finally, the *library manager* component is responsible for maintaining the coherence between related concepts (among them automatically generated *lemmas*) and between the different representations of them in the library (as completely specified terms and as metadata that approximate them).

The actual generation of lemmas is a logic dependent activity that is not directly implemented by the library manager, that is kept logical independent: the component provides hooks to register and invoke logic dependent lemma generators, whose implementation is provided in a component that we describe later and that acts on partially specified terms.

Metadata An extensive library requires an effective and flexible search engine to retrieve concepts. Examples of flexibility are provided by queries up to instantiation or generalization of given formulae, combination of them with extra-logical constraints such as mathematical classification, and retrieval up to minor differences in the matched formula such as permutation of the hypotheses or logical equivalences. Effectiveness is required to exploit the search engine as a first step in automatic tactics. For instance, a paramodulation based procedure must first of all retrieve all the equalities in the distributed library that are likely to be exploited in the proof search. Moreover, since search is mostly logic independent, we would like to implement it on a generic representation of formulae that supports all the previous operations.

In Matita we use relational *metadata* to represent both extra-logical data and a syntactic approximation of a formula (e.g. the constant occurring in head position in the conclusion, the set of constants occurring in the rest of the conclusion and the same information for the hypotheses). The logic dependent *indexing* component, already discussed, generates the syntactic approximation from completely specified terms. The *metadata manager* component stores the metadata in a relational database for scalability and handles, for the library manager, the insertion, removal and indexing of the metadata. The *search engine* component [1] implements the approximated queries on the metadata that can be refined later on, if required, by logic dependent components.

Partially Specified Terms In partially specified terms, subterms can be omitted replacing them with untyped linear placeholders or with typed metavariables (in the style of [8,13]). The latter are Curry-Howard isomorphic to omitted subproofs (conjectures still to be proved).

Completely specified terms are often highly redundant to keep the type-checker simple. This redundant information may be omitted during user-machine communication since it is likely to be automatically inferred by the system replacing conversion with unification [19] in the typing rules (that are relaxed to type inference rules). The *refiner* component of Matita implements unification and the type inference procedure, also inserting implicit coercions [3] to fix local type-checking errors. Coercions are particularly useful in logical systems that lack subtyping [10]. The already discussed library manager is also responsible for the management of coercions, that are constants flagged in a special way.

Subproofs are never redundant and if omitted require tactics to instantiate them with partial proofs that have simpler omitted subterms. Tactics are applied to omitted subterms until the proof object becomes completely specified and can be passed to the library manager. Higher order tactics, usually called tacticals and useful to create more complex tactics, are also implemented in the tactics component. The current implementation in Matita is based on *tinycals* [17], which supports a step-by-step execution of tacticals (usually seen as "black boxes") particularly useful for proof editing, debugging, and maintainability. Tinycals are implemented in Matita in a small but not trivial component that is completely abstracted on the representation of partial proofs.

The *lemma generator* component is responsible for the automatic generation of derived concepts (or lemmas), triggered by the insertion of new concepts in the library. The lemmas are generated automatically computing their statements and then proving them by means of tactics or by direct construction of the proof objects.

Content Level Terms The language used to communicate proofs and especially formulae with the user must also exploit the comfortable and suggestive degree of notational abuse and overloading so typical of the mathematical language. Formalized mathematics cannot hide these ambiguities requiring terms where each symbol has a very precise and definite meaning.

Content level terms provide the (abstract) syntactic structure of the human-oriented (compact, overloaded) encoding. In the *content* component we provide translations from partially specified terms to content level terms and the other way around. The former translation, that loses information, must discriminate between terms used to represent proofs and terms used to represent formulae. Using techniques inspired by [6,7], the formers are translated to a content level representation of proof steps that can in turn easily be rendered in natural language. The representation adopted has greatly influenced the OMDoc [14] proof format that is now isomorphic to it. Terms that represent formulae are translated to MathML Content formulae [12].

The reverse translation for formulae consists in the removal of ambiguity by fixing an interpretation for each ambiguous notation and overloaded symbol used at the content level. The translation is obviously not unique and, if performed locally on each source of ambiguity, leads to a large set of partially specified terms, most of which ill-typed. To solve the problem the *ambiguity manager* component implements an algorithm [18] that drives the translation by alternating translation and refinement steps to prune out ill-typed terms as soon as possible, keeping only the refinable ones. The component is logic independent being completely abstracted over the logical system, the refinement function, and the local translation from content to partially specified terms. The local translation is implemented for occurrences of constants by means of call to the search engine.

The translation from proofs at the content level to partially specified terms is being implemented by means of special tactics following previous work [9,20] on the implementation of declarative proof styles for procedural proof assistants.

Presentation Level Terms Presentation level captures the formatting structure (layout, styles, etc.) of proof expressions and other mathematical entities.

An important difference between the content level language and the presentation level language is that only the former is extensible. Indeed, the presentation level language has a finite vocabulary comprising standard layout schemata (fractions, sub/superscripts, matrices, . . .) and the usual mathematical symbols.

The finiteness of the presentation vocabulary allows its standardization. In particular, for pretty printing of formulae we have adopted MathML Presentation [12], while editing is done using a TEX-like syntax. To visually represent proofs it is enough to embed formulae in plain text enriched with formatting boxes. Since the language of boxes is very simple, many similar specifications exist and we have adopted our own, called BoxML (but we are eager to cooperate for its standardization with other interested teams).

The *notation manager* component provides the translations from content level terms to presentation level terms and the other way around. It also provides a language [15] to associate notation to content level terms, allowing the user to extend the notation used in Matita. The notation manager is logic independent since the content level already is.

The remaining components, mostly logic independent, implement in a modular way the user interface of Matita, that is heavily based on the modern GTK+ toolkit and on standard widgets such as GtkSourceView that implements a programming oriented text editor and GtkMathView that implements rendering of MathML Presentation formulae enabling contextual and controlled interaction with the formula.

The *graph browser* is a GTK+ widget, based on Graphviz, to render dependency graphs with the possibility of contextual interaction with them. It is mainly used in Matita to explore the dependencies between concepts, but other kind of graphs (e.g. the DAG formed by the declared coercions) are also shown.

The *library browser* is a GTK+ window that mimics a web browser, providing a centralized interface for all the searching and rendering functionalities of Matita. It is used to hierarchically browse the library, to render proofs and definitions in natural language, to query the search engine, and to inspect dependency graphs embedding the graph browser.

The *GUI* is the graphical user interface of Matita, inspired by the pioneering work on CtCoq [4] and by Proof General [2]. It differs from Proof General because the sequents are rendered in high quality MathML notation, and because it allows to open multiple library browser windows to interact with the library during proof development.

The hypertextual browsing of the library and proof-by-pointing [5] are both supported by semantic selection. Semantic selection is a technique that consists in enriching the presentation level terms with pointers to the content level terms and to the partially specified terms they correspond to. Highlight of formulae in the widget is constrained to selection of meaningful expressions, i.e. expressions that correspond to a lower level term, that is a content term or a partially or fully specified term. Once the rendering of an upper level term is selected it is possible for the application to retrieve the pointer to the lower level term. An example of applications of semantic selection is *semantic copy & paste*: the user can select an expression and paste it elsewhere preserving its semantics (i.e. the partially specified term), possibly performing some semantic transformation over it (e.g. renaming variables that would be captured or λ-lifting free variables).

Commands to the system can be given either visually (by means of buttons and menus) or textually (the preferred way to input tactics since formulae occurs as tactic arguments). The textual parser for the commands is implemented in the *vernacular* component, that is obviously system (and partially logic) dependent.

To conclude the description of the components of Matita, the *driver* component, which does not act directly on terms, is responsible for pulling together the other components, for instance to parse a command (using the vernacular component) and then triggering its execution (for instance calling the *tactics* component if the command is a tactic).

2.1 Relationship with Other Architectures

An interesting question is which components of Matita have counterparts in systems based on different architectural choices. As an example we consider how we would implement a system based on the following commitments: (1) The architecture is LCF-like. Proof objects are not recorded. (2) The system library is made of scripts. Proof concepts are indexed only after evaluation. (3) The proof language is declarative. Ambiguities in formulae are handled by the type system (e.g. type classes accounts for operator overloading).

Formulae are still represented as presentation, content, partially specified and completely specified terms. Proofs, that are distinct from formulae, exists at the presentation and content level, but do not have a counterpart as partially or completely specified terms. Since only concepts in memory can be queried, metadata are not required: formulae can be indexed using context trees or similar efficient data structures that acts on completely specified formulae.

The following components in Figure 1 have similar counterparts. The *file manager* to store environments obtained processing scripts to avoid re-execution. The *kernel*, that checks definition and theorems, is still present but now it implements the basic tactics, i.e. the tactics that implement reduction and conversion or that correspond to the introduction and elimination rules of the logics. The *indexing* component is not required since in charge of extracting metadata that are neglected. However, the *metadata manager* that used to index metadata is now provided by the context tree manager that indexes the formulae. Logic independence is lost unless the formulae are represented as sort of S-expressions, reintroducing the equivalent of the metadata data type. The *search engine* and the *library manager* are present since the corresponding functionalities (searching and management of derived notions) are still required. All the components that act on partially specified terms are present, even if basic tactics have been moved to the kernel. The *content* component is simplified since the translation (pretty printing) from completely specified terms to content level terms is pointless due to the lack of proof objects. The *ambiguity manager* that acts on content level formulae is removed or it is greatly simplified since type classes take care of overloading. Finally, all the components that act on presentation level terms are present and are likely to be reusable without major changes.

Of course the fact that many components have counterparts among the two set of architectural choices is due to the coarseness of both the description and the provided functionalities. This is wanted. In our opinion the issue of choosing the granularity level of architectures so that smaller components of different systems can be independently compared is non trivial, and was an issue we wanted to address.

3 Functional Architectural Analysis and Reusability

A different classification—other than the data-driven one given in the previous section—of the components shown in Figure 1 is along the lines of the offered functionalities. We grouped the components according to five (macro) functionalities, which are depicted in the vertical partition of Figures 2 and 3: visual interaction and browsing of a mathematical library (*GUI* column), input/output (i.e. parsing and pretty-printing) of formulae and proofs (*I/O* column), indexing and searching of concepts in a library (*search* column), management of a library of certified concepts (*library* column), and interactive development of proofs by means of tactics and decision procedures (*proof authoring* column).

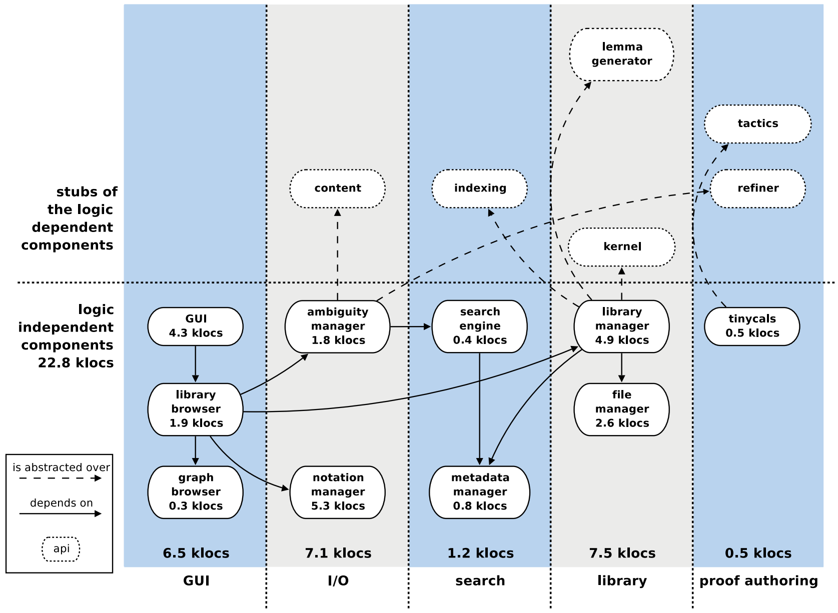

Fig. 2. Logic independent components by functionalities.

In the development history of Matita this classification has been useful for the assignment of development tasks, since the knowledge required for implementing different functionalities varies substantially.

For each functionality it is interesting to assess the degree of coupling of each component with the logical framework. Having a clear separation between the logic dependent components and the logic independent ones should be one of the main guidelines for the development of interactive theorem provers, since it helps to clarify the interface of each component. Moreover, the logic independent functionalities are probably of interest to a broader community.

In Figure 2 we have isolated the logic independent components of Matita (lower part), showing the dependencies among them (solid lines). Some of them depend on "stubs" for logic dependent components, depicted in the upper part of the figure.

The effort for re-targeting Matita to a different logic amounts to provide a new implementation for the stubs. Figure 3 shows the current Matita implementation of CIC. In our case, the logic-dependent components are about 2/3 of the whole code (also due to the peculiar complexity of CIC). However, the real point is that the skills required for implementing the logic-dependent stubs are different from those needed for implementing the logic-independent components, hence potentially permitting to obtain in a reasonable time and with a limited man-power effort a first prototype for early testing. In the next section we investigate this point presenting a detailed timeline for the development of the system.

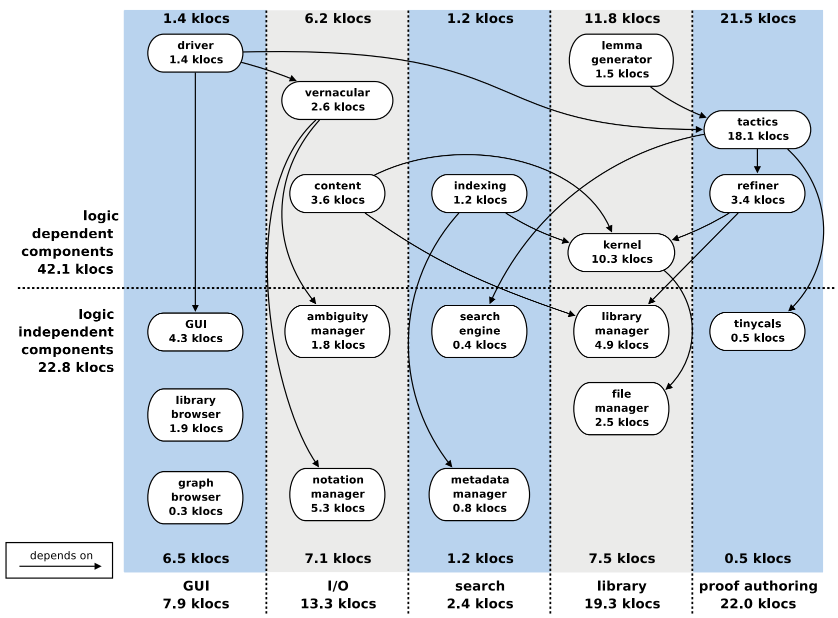

Fig. 3. Stubs implementation for CIC.

A different problem is to understand if there are components that can be reused in systems based on different architectural commitments. A well known example of such a tool is the Proof General generic user interface. Having a reusable user interface is a relatively simple task since not only the user interface is logic independent, but it is also the most external component of a system. We believe that some other logic independent components of Figure 1 could be adapted to other architectures; for instance, components that deal with indexing and searching are likely to be embeddable in any system with minor efforts. This issue of reusability is one of the subject of our current research.

4 System Development

Figure 4 is an hypothetical Gantt-like diagram for the development of an interactive theorem prover with the same architectural commitments of Matita and a logic with comparable complexity. The order in which to develop the components in the figure does not reflect the development history of Matita, where we delayed a few activities, with major negative impacts on the whole schedule.

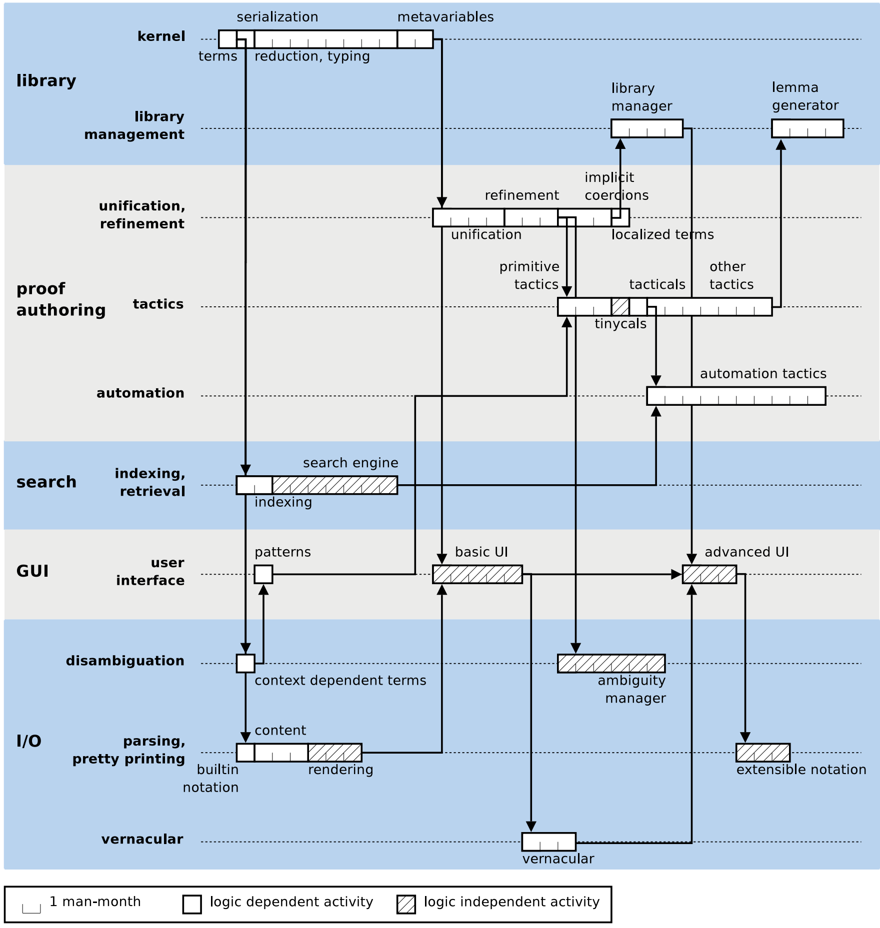

Fig. 4. Gantt-like development schedule of an interactive theorem prover.

The duration of the activities in the diagram is an estimation of the time that would be required now by an independent team to re-implement Matita assuming only the knowledge derivable from the literature.

In any case, in the estimated duration of the activities we are considering the time wasted for rapid prototyping: it is not reasonable in a research community to expect the product to be developed for years without any intermediate prototype to play with. For example, we suggest to implement first reduction and typing in the kernel on completely specified terms before extending it to accommodate metavariables (later on required for partially specified terms). This way the kernel of the type-checker can immediately be tested on a library of concepts exported from another system, and different reduction and type-checking algorithms can be compared leading to possibly interesting research results.

Activities related to logic independent components are marked as dashed in the Gantt-like diagram. If those components are reused in the implementation of the system, most functionalities but interactive proof authoring are made available very early in the development. The bad news are that the overall time required to develop the system will not change, being determined by the complexity of the logic dependent components and their dependencies that limit parallelism. Switching to a simpler logic can probably reduce in a significant way the time required to implement the kernel and the refinement component; however, it is likely to have a minor impact on the time required for tactics and decision procedures. Instead changing the initial architectural commitments (e.g. dropping proof objects and adopting an LCF-like kernel) is likely to change the Gantt in a sensible way. The overall conclusion is that the development of an interactive theorem prover is still a complex job that is unlikely to be substantially simplified in the near future.

The activities of Figure 4 refine the components already presented to improve parallel development and allow rapid prototyping. We describe now the main refinements following the timeline when possible.

We suggest to start developing the kernel omitting support for terms containing metavariables and to add it after the reduction and typing rules for completely specified terms have been debugged. The support for metavariables in the kernel should be kept minimal, only implementing typing rules and unfolding of instantiated metavariables. The core functionalities on partially specified terms, unification and refinement, are implemented in the refiner component outside the kernel. Completely omitting support for metavariables from the kernel is more compliant to the De Bruijn criterion. However, the kernel code for minimal metavariable support is really simple and small, and its omission forces an almost complete re-implementation of the kernel functionalities in the refiner that is better avoided.

Context dependent terms are a necessity for passing to tactics arguments that need to be interpreted (and disambiguated) in a context that is still unknown. In Matita context dependent terms are defined as functions from contexts to terms, but other systems adopt different representations.

Patterns are data structures to represent sequents with selected subterms. They are used as tactic arguments to localize the effect of tactics. Patterns pose a major problem to the design of textual user interfaces, that usually avoid them, but are extremely natural in graphical user interface where they correspond to visual selection (using the mouse) of subterms of the sequent.

A fixed *built-in notation* should be implemented immediately for debugging, followed by the *content* component to map completely (or even partially) specified terms to content and the other way around. Partially specified terms generated by the reverse mapping cannot be processed any further until the refiner component is implemented. Similarly, the reverse mapping of ambiguous terms is delayed until the ambiguity manager is available.

The *rendering* and *extensible notation* activities implement the notation manager component. Initially the machinery to apply extensible notation during rendering is implemented in the rendering activity. A user-friendly language to extend at run time the notation is the subject of the second activity that is better delayed until the interaction mode with the system become clear.

Handling of *implicit coercions* and *localized terms* in the refiner component can be delayed until unification and a light version of refinement are implemented. This way the implementation of tactics can start in advance. Localized terms are data structures to represent partially specified terms obtained by formulae given in input by the user. A refinement error on a localized term should be reported to the user by highlighting (possibly visually) the ill-typed subformula. Localized terms pose a serious problem since several operations such as reduction or insertion of an implicit coercion change or loose the localization information. Thus the refiner must be changed carefully to cope with the two different representations of terms.

The *basic user interface* is an interface to the library that offers browsing, searching and proof-checking, but no tactic based proof authoring. It can, however, already implement proof authoring by direct term manipulation that, once refinement is implemented, can become as advanced as Alf is [11]. The *advanced user interface* offers all the final features of the system, it can be script based and it can present the desired interaction style (procedural versus declarative).

Finally, primitive tactics, that implement the inference rules of the logic, and tacticals are requirements for the development of more advanced interactive tactics and automation tactics, that can proceed in parallel.

5 Conclusions and Future Work

We feel the need for more direct comparisons between different interactive theorem provers to highlight the common functionalities, not only in view of reusability but also to encourage a more systematic comparison of different softwares and architectural solutions. In this paper we have contributed by showing how the architecture of a system (in our case Matita) can be analyzed by classifying its software components along different axes: the representation of formulae and proofs they act on and the macro functionalities they contribute to.

Moreover, we believe that an effort should be made to clearly split the logic dependent components from those that are or can be made logic independent. In addition to be able to clarify the interfaces of the components and their dependencies, this division has immediate applications: having done this for Matita we are now able to clearly estimate the efforts, both in time and in lines of code, required to re-target the system to a different logic. This answers a frequent question posed to us by members of the Types community, since Matita presents technological innovations which are interesting for developers of systems based on logical frameworks other than CIC.

In particular, we have estimated that only one third of the code of Matita (that is still more than 22,000 lines of code) is logic independent, which can be explained by the complexity of the logical system. We have also estimated that re-targeting Matita to a logic with the same complexity as the current one would not require significantly less time than the first implementation (assuming to have enough man power to develop concurrently all parallel tasks). However, working prototypes including even advanced functionalities would be obtained quite early in the development stage, with a positive impact at least on debugging, dissemination, and system evaluation. To our knowledge, similar figures have never been presented for other systems.

A more complex issue is the independence of software components from the main architectural commitments, and consequently their degree of cross-system reusability. We believe that several of the components of Matita have counterparts in systems based on different architectures, and that at least some of them could be embedded in other systems after some modifications. This has already been proven true by Proof General for the graphical user interface. However, that is the easiest case, the graphical user interface being the most external component with no dependencies on it.

Better understanding this issue is one of our current research guidelines, but it requires an initial effort by the whole community to analyze the architectures of several systems according to common criteria in order to identify the corresponding components and to understand how they differ in the various architectures. Our next contribution will consist in a description of the API of the components presented here.

References

1. Andrea Asperti, Ferruccio Guidi, Claudio Sacerdoti Coen, Enrico Tassi, and Stefano Zacchiroli. A content based mathematical search engine: Whelp. In *Post-proceedings of the Types 2004 International Conference*, volume 3839 of *Lecture Notes in Computer Science*, pages 17–32. Springer-Verlag, 2004.
2. David Aspinall. Proof General: A generic tool for proof development. In *Tools and Algorithms for the Construction and Analysis of Systems, TACAS 2000*, volume 1785 of *Lecture Notes in Computer Science*. Springer-Verlag, January 2000.
3. Gilles Barthe. Implicit coercions in type systems. In *Types for Proofs and Programs: International Workshop, TYPES 1995*, pages 1–15, 1995.

4. Yves Bertot. The CtCoq system: Design and architecture. *Formal Aspects of Computing*, 11:225–243, 1999.
5. Yves Bertot, Gilles Kahn, and Laurent Théry. Proof by pointing. In *Symposium on Theoretical Aspects Computer Software (STACS)*, volume 789 of *Lecture Notes in Computer Science*, 1994.
6. Yann Coscoy. *Explication textuelle de preuves pour le Calcul des Constructions Inductives*. PhD thesis, Université de Nice-Sophia Antipolis, 2000.
7. Yann Coscoy, Gilles Kahn, and Laurent Thery. Extracting Text from Proofs. Technical Report RR-2459, Inria (Institut National de Recherche en Informatique et en Automatique), France, 1995.
8. Herman Geuvers and Gueorgui I. Jojgov. Open proofs and open terms: A basis for interactive logic. In J. Bradfield, editor, *Computer Science Logic: 16th International Workshop, CLS 2002*, volume 2471 of *Lecture Notes in Computer Science*, pages 537–552. Springer-Verlag, January 2002.
9. John Harrison. A Mizar Mode for HOL. In Joakim von Wright, Jim Grundy, and John Harrison, editors, *Theorem Proving in Higher Order Logics: 9th International Conference, TPHOLs'96*, volume 1125 of *Lecture Notes in Computer Science*, pages 203–220, Turku, Finland, 1996. Springer-Verlag.
10. Zhaohui Luo. Coercive subtyping. *Journal of Logic and Computation*, 9(1):105–130, 1999.
11. Lena Magnusson and Bengt Nordström. The ALF proof editor and its proof engine. In *Types for Proofs and Programs*, volume 806 of *LNCS*, pages 213–237, Nijmegen, 1994. Springer-Verlag.
12. Mathematical Markup Language (MathML) Version 2.0. W3C Recommendation 21 February 2001, `http://www.w3.org/TR/MathML2`, 2003.
13. César Muñoz. *A Calculus of Substitutions for Incomplete-Proof Representation in Type Theory*. PhD thesis, INRIA, November 1997.
14. OMDoc: An open markup format for mathematical documents (draft, version 1.2). `http://www.mathweb.org/omdoc/pubs/omdoc1.2.pdf`, 2005.
15. Luca Padovani and Stefano Zacchiroli. From notation to semantics: There and back again. In *Proceedings of Mathematical Knowledge Management 2006*, volume 3119 of *Lectures Notes in Artificial Intelligence*, pages 194–207. Springer-Verlag, 2006.
16. Alexandre Riazanov. *Implementing an Efficient Theorem Prover*. PhD thesis, The University of Manchester, 2003.
17. Claudio Sacerdoti Coen, Enrico Tassi, and Stefano Zacchiroli. Tinycals: step by step tacticals. In *Proceedings of User Interface for Theorem Provers 2006*, Electronic Notes in Theoretical Computer Science. Elsevier Science, 2006. To appear.
18. Claudio Sacerdoti Coen and Stefano Zacchiroli. Efficient ambiguous parsing of mathematical formulae. In Andrzej Trybulec Andrea Asperti, Grzegorz Bancerek, editor, *Proceedings of Mathematical Knowledge Management 2004*, volume 3119 of *Lecture Notes in Computer Science*, pages 347–362. Springer-Verlag, 2004.
19. Martin Strecker. *Construction and Deduction in Type Theories*. PhD thesis, Universität Ulm, 1998.
20. Freek Wiedijk. Mmode, a mizar mode for the proof assistant coq. Technical Report NIII-R0333, University of Nijmegen, 2003.