Brokers and Web-Services for Automatic Deduction: a Case Study

Claudio Sacerdoti Coen* and Stefano Zacchiroli**

[1] Department of Computer Science
University of Bologna
Mura Anteo Zamboni 7, 40127 Bologna, ITALY
sacerdot@cs.unibo.it
[2] Department of Computer Science
École Normale Supérieure
45, Rue d'Ulm, F-75230 Paris Cedex 05, FRANCE
zack@cs.unibo.it

Abstract. We present a planning broker and several Web-Services for automatic deduction. Each Web-Service implements one of the tactics usually available in interactive proof-assistants. When the broker is submitted a "proof status" (an incomplete proof tree and a focus on an open goal) it dispatches the proof to the Web-Services, collects the successful results, and send them back to the client as "hints" as soon as they are available.
In our experience this architecture turns out to be helpful both for experienced users (who can take benefit of distributing heavy computations) and beginners (who can learn from it).

1 Introduction

The Web-Service approach at software development seems to be a working solution for getting rid of a wide range of incompatibilities between communicating software applications. W3C's efforts in standardizing related technologies grant longevity and implementations availability for frameworks based on Web-Services for information exchange. As a direct consequence, the number of such frameworks is increasing and the World Wide Web is moving from a disorganized repository of human-understandable HTML documents to a disorganized repository of applications working on machine-understandable XML documents both for input and output.

The big challenge for the next future is to provide stable and reliable services over this disorganized, unreliable, and ever-evolving architecture. The standard solution is to provide a further level of stable services (called *brokers*) that behave

* Partially supported by 'MoWGLI: Math on the Web, Get it by Logic and Interfaces', EU IST-2001-33562
** Partially supported by 'MyThS: Models and Types for Security in Mobile Distributed Systems', EU FET-GC IST-2001-32617

as common gateways/addresses for client applications to access a wide variety of services and abstract over them.

Since the *Declaration of Linz*, the MONET Consortium[1] is working on the development of a framework, based on the Web-Services/brokers approach, aimed at providing a set of software tools for the advertisement and the discovery of mathematical Web-Services.

Several groups have already developed software bus and services[2] providing both computational and reasoning capabilities [3, 4, 15, 16]: the first ones are implemented on top of Computer Algebra Systems; the second ones provide interfaces to well-known theorem provers. Proof-planners, proof-assistants, CASs and domain-specific problem solvers are natural candidates to be clients of these services. Nevertheless, so far the number of examples in the literature has been insufficient to fully assess the concrete benefits of the framework.

In this paper we present an architecture, namely H-Bugs, implementing a *suggestion engine* for the proof assistant developed on behalf of the HELM[3] project [5]. We provide several Web-Services (called *tutors*) able to suggest possible ways to proceed in a proof. The tutors are orchestrated by a broker (a Web-Service itself) that is able to dispatch a proof status from a client (the proof-assistant) to the tutors; each tutor tries to make progress in the proof and, in case of success, notifies the client that shows an *hint* to the user. The broker is an instance of the homonymous entity of the MONET framework. The tutors are MONET services. Another Web-Service (which is not described in this paper and which is called Getter [14]) is used to locate and download mathematical entities; the Getter plays the role of the Mathematical Object Manager of the MONET framework.

A precursor of H-Bugs is the Ωmega-Ants project [6, 7], which provided similar functionalities to the Ωmega proof-planner [8]. The main architectural difference between H-Bugs and Ωmega-Ants is that the latter is based on a blackboard architecture and it is not implemented using Web-Services and brokers.

In Sect. 2 we present the architecture of H-Bugs. A usage session is shown in Sect. 3. Further implementation details are given in Sect. 4. Sect. 5 is an overview of the tutors that have been implemented. As usual, the final section of this paper is devoted to conclusions and future works.

2 An H-Bugs Bird's Eye View

The H-Bugs architecture (depicted in Fig. 1) is based on three different kinds of actors: *clients*, *brokers*, and *tutors*. Each actor presents one or more Web-Service interfaces to its neighbors H-Bugs actors.

---

[1] http://monet.nag.co.uk/cocoon/monet/index.html

[2] The most part of these systems predate the development of Web-Services. Those systems whose development is still active are slowly being reimplemented as Web-Services.

[3] Hypertextual Electronic Library of Mathematics, http://helm.cs.unibo.it

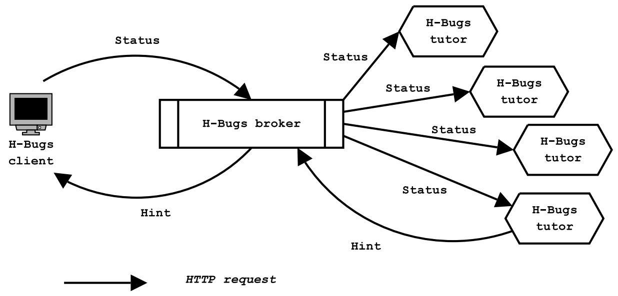

Fig. 1. H-Bugs architecture

In this section we detail the role and requirements of each kind of actors and we discuss about the correspondences between them and the MONET entities described in [11]. Due to lack of space, we cannot compare our framework to similar proposals, as the older and more advanced Ωmega system. The study of the correspondences with MONET is well motivated by the fact that the MONET framework is still under development and that our implementation is one of the first experiments in Web-Servicebased distributed reasoning. On the other hand, a comparison with Ωmega would be less interesting since the functionalities we provide so far are just a subset of the Ωmega-Ants ones.

Clients An H-Bugs client is a software component able to produce *proof status* and to consume *hints*.

A proof status is a representation of an incomplete proof and is supposed to be informative enough to be used by an interactive proof assistant. No additional requirements exist on the proof status, but there should be an agreement on its format between clients and tutors. A hint is an encoding of a step that can be performed in order to proceed in an incomplete proof. Usually it represents a reference to a tactic available on some proof assistant along with an instantiation for its formal parameters. Hints can also be more structured: a hint can be as complex as a whole proof-plan.

Using W3C's terminology [1], clients act both as Web-Service providers and requesters, see Fig. 2. They act as providers receiving hints from the broker; they act as requesters submitting new status to the tutors. Clients additionally use broker services to know which tutors are available and to subscribe to one or more of them.

Usually, when the client role is taken by an interactive proof assistant, new status are sent to the broker as soon as the proof change (e.g. when the user applies a tactic or when a new proof is started); hints are shown to the user by the means of some effects in the user interface (e.g. popping a dialog box or enlightening a tactic button).

H-Bugs clients act as MONET clients and ask brokers to provide access to a set of services (the tutors). H-Bugs has no actors corresponding to MONET's

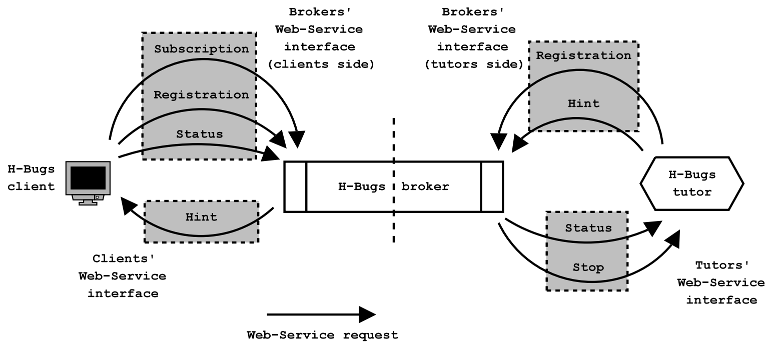

Fig. 2. H-Bugs Web-Services interfaces

Broker Locating Service (since the client is supposed to know the URI of at least one broker). The H-Bugs clients and tutors contact the Getter (a MONET Mathematical Object Manager) to locate and retrieve mathematical items from the HELM library. The proof status that are exchanged by the H-Bugs actors, instead, are built on the fly and are neither stored nor given an unique identifier (URI) to be managed by the Getter.

Brokers Brokers are the key actors of the H-Bugs architecture since they act as intermediaries between clients and tutors. They behave as Web-Services providers and requesters for *both* clients and tutors, see Fig. 2.

With respect to the client, a broker acts as a Web-Service provider, receiving the proof status and forwarding it to one or more tutors. It also acts as a Web-Service requester sending hints to the client as soon as they are available from the tutors.

With respect to the tutors, the Web-Service provider role is accomplished by receiving hints as soon as they are produced; as a requester, it is accomplished by asking for computations (*musings* in H-Bugs terminology) on status received by clients and by stopping already late but still ongoing `musings`.

Additionally brokers keep track of available tutors and clients subscriptions.

H-Bugs brokers act as MONET brokers implementing the following components: Client Manager, Service Registry Manager (keeping track of available tutors), Planning Manager (choosing the available tutors among the ones to which the client is subscribed), Execution Manager. The Service Manager component is not required since the session handler, that identifies a session between a service and a broker, is provided to the service by the broker instead of being received from the service when the session is initialized. In particular, a session is identified by an unique identifier for the client (its URL) and an unique identifier for the broker (its URL).

Notice that H-Bugs brokers have no knowledge of the domain area of proof-assistants, nor they are able to interpret the messages that they are forwarding.

They are indeed only in charge of maintaining the abstraction of several reasoning blackboards — one for each client — of capacity one: a blackboard is created when the client submits a problem; it is then "shared" by the client and all the tutors until the client submits the next problem. For instance, replacing the client with a CAS and all the tutors with agents implementing different resolution methods for differential equations would not require any change in the broker. Notice that all the tutors must expose the same interface to the broker.

The MONET architecture specification does not state explicitly whether the service and broker answers can be asynchronous. Nevertheless, the described information flow implicitly suggests a synchronous implementation. On the contrary, in H-Bugs every request is asynchronous: the connection used by an actor to issue a query is immediately closed; when a service produces an answer, it gives it back to the issuer by calling the appropriate actor's method.

Tutors Tutors are software components able to consume proof status producing hints. H-Bugs does not specify by which means hints should be produced: tutors can use any means necessary (heuristics, external theorem prover or CAS, etc.). The only requirement is that there exists an agreement on the formats of proof status and hints.

Tutors act both as Web-Service providers and requesters for the broker, see Fig. 2. As providers, they wait for commands requesting to start a new `musing` on a given proof status or to stop an old, out of date, `musing`. As requesters, they signal to the broker the end of a `musing` along with its outcome (a hint in case of success or a failure notification).

H-Bugs tutors act as MONET services.

3 An H-Bugs Session Example

In this section we describe a typical H-Bugs session. The aim of the session is to solve the following easy exercise:

Exercise 1. Let x be a generic real number. Using the HELM proof-engine, prove that

$$x = \frac{(x+1)*(x+1)-1-x*x}{2}$$

Let us suppose that the H-Bugs broker is already running and that the tutors already registered themselves to the broker. When the user starts our proof-engine `gTopLevel`, the system registers itself to the broker, that sends back the list of available tutors. By default, `gTopLevel` notifies to the broker its intention of subscribing to every tutor available. The user can always open a configuration window where she is presented the list of available tutors and she can independently subscribe and unsubscribe herself to each tutor.

The user can now insert into the system the statement of the theorem and start proving it. Let us suppose that the first step of the user is proving that the denominator 2 is different from 0. Once that this technical result is proven,

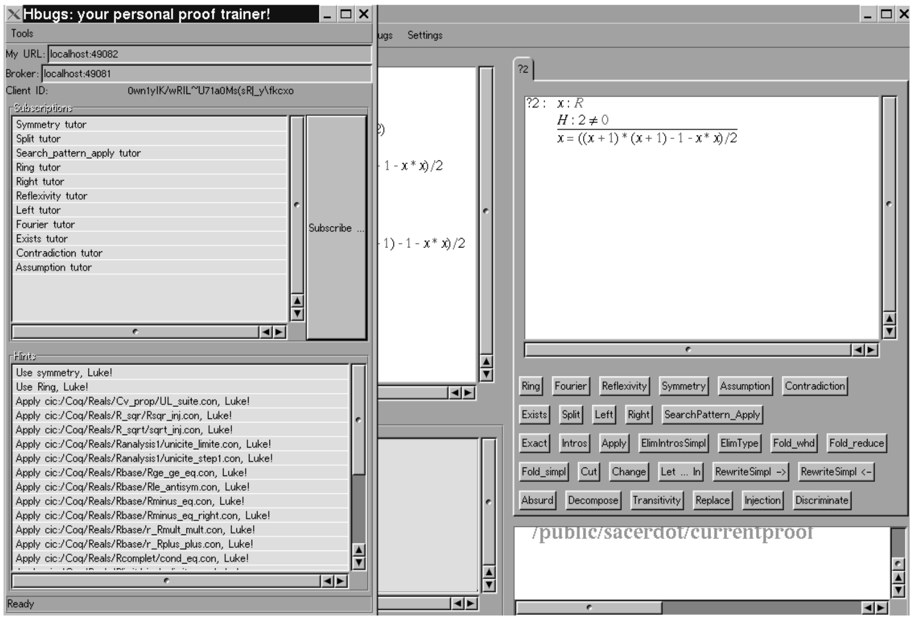

Fig. 3. Example session.

the user must prove the goal shown in the upper right corner of the window in background in Fig. 3.

While the user is wondering how to proceed in the proof, the tutors are trying to progress in the proof. After a while, the tutors' suggestions start to appear in the lower part of the H-Bugs interface window (the topmost window in Fig. 3). In this case, the tutors are able to produce 23 hints. The first and not very useful hint suggests to proceed in the proof by exchanging the two sides of the equality. The second hint suggests to reduce both sides of the equality to their normal form by using only reductions which are justified by the ring structure of the real numbers; the two normal forms, though, are so different that the proof is not really simplified. All the residual 21 hints suggest to apply one lemma from the distributed library of HELM. The user can look at the statement of any lemma by clicking on its URI.

The user can now look at the list of suggestions and realize that a good one is applying the lemma `r_Rmult_mult` that allows to multiply both equality members by the same scalar[4]. Double-clicking on the hint automatically applies the lemma, reducing the proof to closing three new goals. The first one asks the user the scalar to use as an argument of the previous lemma; the second one

[4] Even if she does not receive the hint, the user probably already knows that this is the right way to proceed. The difficult part, accomplished by the hint, is guessing what is the name of the lemma to apply.

states that the scalar is different from 0; the third lemma (the main one) asks to prove the equality between the two new members.

The user proceeds by instantiating the scalar with the number 2. The `Assumption` tutor now suggests to close the second goal (that states that $2 \neq 0$) by applying the hypothesis H. No useful suggestions, instead, are generated for the main goal $2*x = 2*((x+1)*(x+1)-1-x*x)*2^{-1}$. To proceed in the proof the user needs to simplify the expression using the lemma *Rinv_r_simpl_m* that states that $\forall x, y.\, y = x*y*x^{-1}$. Since we do not provide yet any tutor suggesting simplifications, the user must find out this simplification by himself. Once she founds it, the goal is reduced to proving that $2*x = (x+1)*(x+1)-1-x*x$. This equality is easily solved by the `Ring` tutor, that suggests[5] to the user how to complete the proof in one macrostep.

4 Implementation's Highlights

In this section we present some of the most relevant implementation details of the H-Bugs architecture.

Proof status In our implementation of the H-Bugs architecture we used the proof assistant of the HELM project (codename `gTopLevel`) as an H-Bugs client. Thus we have implemented serialization/deserialization capabilities for its internal status. In order to be able to describe Web-Services that exchange status in WSDL using the XML Schema type system, we have chosen an XML format as the target format for the serialization.

Each proof is represented by a tuple of four elements: *uri*, *metasenv*, *proof*, *thesis*.

uri : an URI chosen by the user at the beginning of the proof process. Once (and if) proved, that URI will globally identify the term inside the HELM library (given that the user decides to save it).

thesis : the ongoing proof thesis

proof : the current incomplete proof tree. It can contain *metavariables* (holes) that stands for the parts of the proof that are still to be completed. Each metavariable appearing in the tree references one element of the metavariables environment (*metasenv*).

metasenv : the metavariables environment is a list of *goals* (unproved conjectures). In order to complete the proof, the user has to instantiate every metavariable in the proof with a proof of the corresponding goal. Each goal is identified by an unique identifier and has a context and a type (the goal thesis). The context is a list of named hypotheses (declarations and definitions). Thus the context and the goal thesis form a sequent, which is the statement of the proof that will be used to instantiate the metavariable occurrences.

[5] The `Ring` suggestion is just one of the 22 hints that the user receives. It is the only hint that does not open new goals, but the user right now does not have any way to know that.

Each of these information is represented in XML as described in [12]. Additionally, an H-Bugs status carries the unique identifier of the current goal, which is the goal the user is currently focused on. Using this value it is possible to implement different client side strategies: the user could ask the tutors to work on the goal she is considering or to work on the other "background" goals.

Hints A hint in the H-Bugs architecture should carry enough information to permit the client to progress in the current proof. In our implementation each hint corresponds to either one of the tactics available to the user in gTopLevel (together with its actual arguments) or a set of alternative suggestions (a list of hints).

For tactics that do not require any particular argument (like tactics that apply type constructors or decision procedures) only the tactic name is represented in the hint. For tactics that need terms as arguments (for example the `Apply` tactic that apply a given lemma) the hint includes a textual representation of them, using the same representation used by the interactive proof assistant when querying user for terms. In order to be transmitted between Web-Services, hints are serialized in XML.

It is also possible for a tutor to return more hints at once, grouping them in a particular XML element. This feature turns out to be particularly useful for the *searchPatternApply* tutor (see Sect. 5) that queries a lemma database and returns to the client a list of all lemmas that could be used to complete the proof. This particular hint is encoded as a list of `Apply` hints, each of them having one of the results as term argument.

We would like to stress that the H-Bugs architecture has no dependency on either the hint or the status representation: the only message parts that are fixed are those representing the administrative messages (the envelopes in the Web-Services terminology). In particular, the broker can manage at the same time several sessions working on different status/hints formats. Of course, there must be an agreement between the clients and the tutors on the format of the data exchanged.

In our implementation the client does not trust the tutors hints: being encoded as references to available tactics imply that an H-Bugs client, at the receipt of a hint, simply try to replay the work done by a tutor on the local copy of the proof. The application of the hint can even fail to type check and the client copy of the proof can be left undamaged after spotting the error. Note, however, that it is still possible to implement a complex tutor that looks for a proof doing backtracking and that send back to the client a hint whose argument is a witness (a trace) of the proof found: the client applies the hint reconstructing (and checking the correctness of) the proof from the witness, without having to re-discover the proof itself.

An alternative implementation where the tutors are trusted would simply send back to the client a new proof-status. Upon receiving the proof-status, the client would just override its current proof status with the suggested one. In the case of those clients which are implemented using proof-objects (as the Coq proof-assistant, for instance), it is still possible for the client to type-check the proof-object and reject wrong hints. The systems that are not based on proof-objects (as PVS, NuPRL, etc.), instead, must completely trust the new proof-status. In this case the H-Bugs architecture would need at least to be extended with clients-tutors authentication.

Registries Being central in the H-Bugs architecture, the broker is also responsible of housekeeping operations both for clients and tutors. These operations are implemented using three different data structures called *registries*: clients registry, tutors registry and `musings` registry.

In order to use the suggestion engine a client should register itself to the broker and subscribe to one or more tutors. The registration phase is triggered by the client using the `Register_client` method of the broker to send him an unique identifier and its base URI as a Web-Service. After the registration, the client can use the `List_tutors` method of the broker to get a list of available tutors. Eventually the client can subscribe to one or more of these using the `Subscribe` method of the broker. Clients can also unregister from brokers using `Unregister_client` method.

The broker keeps track of both registered clients and clients' subscriptions in the clients registry.

In order to be advertised to clients during the subscription phase, tutors should register to the broker using the `Register_tutor` method of the broker. This method is really similar to `Register_client`: tutors are required to send an unique identifier and a base URI for their Web-Service. Additionally tutors are required to send an human readable description of their capabilities; this information could be used by the client user to decide which tutors she wants to subscribe to. As the clients, tutors can unregister from brokers using `Unregister_broker` method.

Each time the client status changes, it get sent sent to the broker using its `Status` method. Using both the clients registry (to lookup the client's subscription) and the tutors registry (to check if some tutors have unsubscribed), the broker is able to decide to which tutors the new status have to be forwarded.

The forwarding operation is performed using the `Start_musing` method of the tutors, that is a request to start a new computation (*`musing`*) on a given status. The return value of `Start_musing` is a `musing` identifier that is saved in the `musings` registry along with the identifier of the client that triggered the `musing`.

As soon as a tutor completes an `musing`, it informs the broker using its `Musing_completed` method; the broker can now remove the `musing` entry from the `musings` registry and, depending on its outcome, inform the client. In case of success one of the `Musing_completed` arguments is a hint to be sent to the client; otherwise there is no need to inform him and the `Musing_completed` method is called just to update the `musings` registry.

Consulting the `musings` registry, the broker is able to know, at each time, which `musings` are in execution on which tutor. This peculiarity is exploited by the broker on invocation of the `Status` method. Receiving a new status from the client implies indeed that the previous status no longer exists and all `musings` working on it should be stopped: additionally to the already described behavior (i.e. starting new `musings` on the received status), the broker takes also care of stopping ongoing computation invoking the `Stop_musing` method of the tutors.

Tutors Each tutor exposes a Web-Service interface and should be able to work, not only for many different clients referring to a common broker, but also for many different brokers. The potential high number of concurrent clients imposes a multi-threaded or multi-process architecture.

Our current implementation is based on a multi threaded architecture exploiting the capabilities of the O'HTTP library [14]. Each tutor is composed by one always running thread plus an additional thread for each `musing`. One thread is devoted to listening for incoming Web-Service requests; when a request is received the control is passed to a second thread, created on the fly, that handle the incoming request (usual one-thread-per-request approach in web servers design). In particular if the received request is `Start_musing`, a new thread is spawned to handle it; the thread in duty to handle the HTTP request returns an HTTP response containing the identifier of the just started `musing`, and then dies. If the received request is `Stop_musing`, instead, the spawned thread kills the thread responsible for the `musing` whose identifier is the argument of the `Stop_musing` method.

This architecture turns out to be scalable and allows the running threads to share the cache of loaded (and type-checked) theorems. As we will explain in Sect. 5, this feature turns out to be really useful for tactics that rely on a huge but fixed set of lemmas, as every reflexive tactic.

The implementation of a tutor within the described architecture is not that difficult having a language with good threading capabilities (as OCaml has) and a pool of already implemented tactics (as `gTopLevel` has). Working with threads is known to be really error prone due to concurrent programming intrinsic complexity. Moreover, there is a non-neglectable part of code that needs to be duplicated in every tutor: the code to register the tutor to the broker and to handle HTTP requests; the code to manage the creation and termination of threads; and the code for parsing the requests and serializing the answers. As a consequence we have written a generic implementation of a tutor which is parameterized over the code that actually proposes the hint and over some administrative data (as the port the tutor will be listening to).

The generic tutor skeleton is really helpful in writing new tutors. Nevertheless, the code obtained by converting existing tactics into tutors is still quite repetitive: every tutor that wraps a tactic has to instantiate its own copy of the proof-engine kernel and, for each request, it has to override its status, guess the tactic arguments, apply the tactic and, in case of success, send back a hint with the tactic name and the chosen arguments. Of course, the complex part of the work is guessing the right arguments. For the simple case of tactics that do not require any argument, though, we are able to automatically generate the whole tutor code given the tactic name. Concretely, we have written a tactic-based tutor template and a script that parses an XML file with the specification of the tutor and generates the tutor's code. The XML file describes the tutor's port, the code to invoke the tactic, the hint that is sent back upon successful application and a human readable explanation of the tactic implemented by the tutor.

5 The Implemented H-BugsTutors

To test the H-Bugs architecture and to assess the utility of a suggestion engine for the end user, we have implemented several tutors. In particular, we have investigated three classes of tutors:

1. *Tutors for beginners*. These are tutors that implement tactics which are neither computationally expensive nor difficult to understand: an expert user can always understand if the tactic can be applied or not without having to try it. For example, the following implemented tutors belong to this class:
   - *Assumption Tutor*: it ends the proof if the thesis is equivalent[6] to one of the hypotheses[7].
   - *Contradiction Tutor*: it ends the proof by *reductio ad adsurdum* if one hypothesis is equivalent to $False$.
   - *Symmetry Tutor*: if the goal thesis is an equality, it suggests to apply the commutative property.
   - *Left/Right/Exists/Split/Reflexivity/Constructor Tutors*: the Constructor Tutor suggests to proceed in the proof by applying one or more constructors when the goal thesis is an inductive type or a proposition inductively defined according to the declarative style[8]. Since disjunction, conjunction, existential quantification and Leibniz equality are particular cases of inductive propositions, all the other tutors of this class are instantiations of the the Constructor tactic. Left and Right suggest to prove a disjunction by proving its left/right member; Split reduces the proof of a conjunction to the two proof of its members; Exists suggests to prove an existential quantification by providing a witness[9]; Reflexivity proves an equality whenever the two sides are convertible.

[6] In our implementation, the equivalence relation imposed by the logical framework is *convertibility*. Two expressions are convertible when they reduce to the same normal form. Two "equal" terms depending on free variables can be non-convertible since free variables stop the reduction. For example, $2x$ is convertible with $(3-1)x$ because they both reduce to the same normal form $x+x+0$; but $2x$ is not convertible to $x2$ since the latter is already in normal form.

[7] In some cases, especially when non-trivial computations are involved, the user is totally unable to figure out the convertibility of two terms. In these cases the tutor becomes handy also for expert users.

[8] An example of a proposition that can be given in declarative style is the $\leq$ relation over natural numbers: $\leq$ is the smallest relation such that $n \leq n$ for every n and $n \leq m$ for every n, m such that $n \leq p$ where p is the predecessor of m. Thus, a proof of $n \leq n$ is simply the application of the first constructor to n and a proof of $n \leq m$ is the application of the second constructor to n, m and a proof of $n \leq m$.

[9] This task is left to the user.

Beginners, when first faced with a tactic-based proof-assistant, get lost quite soon since the set of tactics is large and their names and semantics must be remembered by heart. Tutorials are provided to guide the user step-by-step in a few proofs, suggesting the tactics that must be used. We believe that our beginners tutors can provide an auxiliary learning tool: after the tutorial, the user is not suddenly left alone with the system, but she can experiment with variations of the exercises given in the tutorial as much as she like, still getting useful suggestions. Thus the user is allowed to focus on learning how to do a formal proof instead of wasting efforts trying to remember the interface to the system.

2. *Tutors for Computationally Expensive Tactics.* Several tactics have an unpredictable behavior, in the sense that it is unfeasible to understand whether they will succeed or they will fail when applied and what will be their result. Among them, there are several tactics either computationally expensive or resource consuming. In the first case, the user is not willing to try a tactic and wait for a long time just to understand its outcome: she would prefer to keep on concentrating on the proof and have the tactic applied in background and receive out-of-band notification of its success. The second case is similar, but the tactic application must be performed on a remote machine to avoid overloading the user host with several concurrent resource consuming applications.
Finally, several complex tactics and in particular all the tactics based on reflexive techniques depend on a pretty large set of definitions, lemmas and theorems. When these tactics are applied, the system needs to retrieve and load all the lemmas. Pre-loading all the material needed by every tactic can quickly lead to long initialization times and to large memory footstamps. A specialized tutor running on a remote machine, instead, can easily pre-load the required theorems.
As an example of computationally expensive task, we have implemented a tutor for the *Ring* tactic [9]. The tutor is able to prove an equality over a ring by reducing both members to a common normal form. The reduction, which may require some time in complex cases, is based on the usual commutative, associative and neutral element properties of a ring. The tactic is implemented using a reflexive technique, which means that the reduction trace is not stored in the proof-object itself: the type-checker is able to perform the reduction on-the-fly thanks to the conversion rules of the system. As a consequence, in the library there must be stored both the algorithm used for the reduction and the proof of correctness of the algorithm, based on the ring axioms. This big proof and all of its lemmas must be retrieved and loaded in order to apply the tactic. The Ring tutor loads and caches all the required theorems the first time it is contacted.
3. *Intelligent Tutors.* Expert users can already benefit from the previous class of tutors. Nevertheless, to achieve a significative production gain, they need more intelligent tutors implementing domain-specific theorem provers or able to perform complex computations. These tutors are not just plain implementations of tactics or decision procedures, but can be more complex software

agents interacting with third-parties software, such as proof-planners, CAS or theorem-provers.
To test the productivity impact of intelligent tutors, we have implemented a tutor that is interfaced with the HELM Search-Engine[10] and that is able to look for every theorem in the distributed library that can be applied to proceed in the proof. Even if the tutor deductive power is extremely limited[11], it is not unusual for the tutor to come up with precious hints that can save several minutes of work that would be spent in proving again already proven results or figuring out where the lemmas could have been stored in the library.

6 Conclusions and Future Work

In this paper we described a suggestion engine architecture for proof-assistants: the client (a proof-assistant) sends the current proof status to several distributed Web-Services (called tutors) that try to progress in the proof and, in case of success, send back an appropriate hint (a proof-plan) to the user. The user, that in the meantime was able to reason and progress in the proof, is notified with the hints and can decide to apply or ignore them. A broker is provided to decouple the clients and the tutors and to allow the client to locate and invoke the available remote services. The whole architecture is an instance of the MONET architecture for Mathematical Web-Services. It constitutes a reimplementation of the core features of the pioneering Ωmega-Ants system in the new Web-Services framework.

A running prototype has been implemented as part of the HELM project [5] and we already provide several tutors. Some of them are simple tutors that try to apply one or more tactics of the HELM Proof-Engine, which is also our client. We also have a much more complex tutor that is interfaced with the HELM Search-Engine and looks for lemmas that can be directly applied.

Future works comprise the implementation of new features and tutors, and the embedding of the system in larger test cases. For instance, one interesting case study would be interfacing a CAS as Maple to the H-Bugs broker, developing at the same time a tutor that implements the Field tactic of Coq, which proves the equality of two expressions in an abstract field by reducing both members to the same normal form. CASs can produce several compact normal forms, which are particularly informative to the user and that may suggest how to proceed in a proof. Unfortunately, CASs do not provide any certificate about the correctness of the simplification. On the contrary, the Field tactic certifies the equality of two expressions, but produces normal forms that are hardly a simplification of the original formula. The benefits for the CAS would be obtained by using the Field tutor to certify the CAS simplifications, proving that the Field normal form of an expression is preserved by the simplification. More advanced tutors

[10] http://helm.cs.unibo.it/library.html

[11] We do not attempt to check if the new goals obtained applying a lemma can be automatically proved or, even better, automatically disproved to reject the lemma.

could exploit the CAS to reduce the goal to compact normal forms [10], making the Field tutor certify the simplification according to the skeptical approach.

We have many plans for further developing both the H-Bugs architecture and our prototype. Interesting results could be obtained augmenting the informative content of each suggestion. We can for example modify the broker so that also negative results are sent back to the client. Those negative suggestions could be reflected in the user interface by deactivating commands to narrow the choice of tactics available to the user. This approach could be interesting especially for novice users, but requires the client to increase their level of trust in the other actors.

We plan also to add some rating mechanism to the architecture. A first improvement in this direction could be distinguishing between hints that, when applied, are able to completely close one or more goals, and tactics that progress in the proof by reducing one or more goals to new goals: since the new goals can be false, the user can be forced later on to backtrack.

Other heuristics and or measures could be added to rate hints and show them to the user in a particular order: an interesting one could be a measure that try to minimize the size of the generated proof, privileging therefore non-overkilling solutions [13].

We are also considering to follow the Ωmega-Ants path adding "recursion" to the system so that the proof status resulting from the application of old hints are cached somewhere and could be used as a starting point for new hint searches. The approach is interesting, but it represents a big shift towards automatic theorem proving: thus we must consider if it is worth the effort given the increasing availability of automation in proof assistants tactics and the ongoing development of Web-Services based on already existent and well developed theorem provers.

Even if not strictly part of the H-Bugs architecture, the graphical user interface (GUI) of our prototype needs a lot of improvement if we want it to be really usable by novices. In particular, a critical issue is avoiding continuous distractions for the user determined by the hints that are asynchronously pushed to her.

Our Web-Services still lack a real integration in the MONET architecture, since we do not provide the different ontologies to describe our problems, solutions, queries, and services. In the short term, completing this task could provide a significative feedback to the MONET consortium and would enlarge the current set of available MONET actors on the Web. In the long term, new more intelligent tutors could be developed on top of already existent MONET Web-Services.

To conclude, H-Bugs is a nice experiment meant to understand whether the current Web-Services technology is mature enough to have a concrete and useful impact on the daily work of proof-assistants users. So far, only the tutor that is interfaced with the HELM Search-Engine has effectively increased the productivity of experts users. The usefulness of the tutors developed for beginners, instead, need further assessment.

References

1. Web Services Glossary, W3C Working Draft, 14 May 2003.
http://www.w3.org/TR/2003/WD-ws-gloss-20030514/
2. Web Services Description Language (WSDL) Version 1.2: Bindings, W3C Working Draft, 24 January 2003.
http://www.w3.org/TR/wsdl12-bindings/
3. A. Armando, D. Zini. Interfacing Computer Algebra and Deduction Systems via the Logic Broker Architecture. In Proceedings of the Eighth Calculemus symphosium, St. Andrews, Scotland, 6–7 August 2000.
4. O. Caprotti. Symbolic Evaluator Service. Project Report of the MathBrocker Project, RISC-Linz, Johannes Kepler University, Linz, Austria, May 2002.
5. A. Asperti, F. Guidi, L. Padovani, C. Sacerdoti Coen, I. Schena. Mathematical Knowledge Management in HELM. In Annals of Mathematics and Artificial Intelligence, 38(1): 27–46, May 2003.
6. C. Benzmüller, V. Sorge. O-Ants – An Open Approach at Combining Interactive and Automated Theorem Proving. In M. Kerber and M. Kohlhase (eds.), Integration of Symbolic and Mechanized Reasoning, pp. 81–97, 2000.
7. C. Benzmüller, M. Jamnik, M. Kerber, V. Sorge. Agent-based Mathematical Reasoning. In A. Armando and T. Jebelean (eds.), Electronic Notes in Theoretical Computer Science, (1999) 23(3), Elsevier.
8. C. Benzmüller, L. Cheikhrouhou, D. Fehrer, A. Fiedler, X. Huang, M. Kerber, M. Kohlhase, K. Konrad, E. Melis, A. Meier, W. Schaarschmidt, J. Siekmann, V. Sorge. OMEGA: Towards a Mathematical Assistant. In W. McCune (ed), Proceedings of the 14th Conference on Automated Deduction (CADE-14), Springer LNAI vol. 1249, pp. 252–255, Townsville, Australia, 1997.
9. S. Boutin. Using reflection to build efficient and certified decision procedures. In Martin Abadi and Takahashi Ito, editors, TACS'97, volume 1281. LNCS, Springer-Verlag, 1997.
10. David Delahaye, Micaela Mayero. A Maple Mode for Coq. Contribution to the Coq library.
htpp://coq.inria.fr/contribs/MapleMode.html
11. The MONET Consortium, MONET Architecture Overview, Public Deliverable D04 of the MONET Project.
http://monet.nag.co.uk/cocoon/monet/publicsdocs/monet-overview.pdf
12. C. Sacerdoti Coen. Exportation Module, MoWGLI Deliverable D2.a.
http://mowgli.cs.unibo.it/html_no_frames/deliverables/transformation/d2a.html
13. C. Sacerdoti Coen. Tactics in Modern Proof-Assistants: the Bad Habit of Overkilling. In Supplementary Proceedings of the 14th International Conference TPHOLS 2001, pp. 352–367, Edinburgh.
14. S. Zacchiroli. *Web services per il supporto alla dimostrazione interattiva*, Master Thesis, University of Bologna, 2002.
15. J. Zimmer and M. Kohlhase. System Description: The MathWeb Software Bus for Distributed Mathematical Reasoning. In Proceedings of the 18th International Conference on Automated Deduction CADE 18, LNAI 2392, Springer Verlag, 2002.
16. R. Zippel. The MathBus. In Workshop on Internet Accessible Mathematical Computation at ISSAC'99, Vancouver, Canada, July 28–31, 1999.