This is a revised and extended version for journal publication of a paper in *Proceedings, International Symposium on Multimedia Software Engineering*, ed. Jeffrey Tsai and Po-Jen Chuang, IEEE Press, pages 25–32, 2000; the symposium was held in Taipai, Taiwan, in December 2000.

Web-based Support for Cooperative Software Engineering[1]

Joseph A. Goguen and Kai Lin
Department of Computer Science & Engineering
University of California at San Diego, La Jolla CA 92093-0114 USA

Abstract The Tatami project is building a system to support software engineering over the internet, exploiting recent advances in web technology, interface design, and specification. Our effort to improve the usability of such systems has led us into algebraic semiotics, while our effort to develop better formal methods for distributed concurrent systems has led us into hidden algebra and fuzzy logic. This paper discusses the Tatami system design, especially its software architecture, and its user interface principles. New work in the latter area includes an extension of algebraic semiotics to dynamic multimedia interfaces, and integrating Gibsonian affordances with algebraic semiotics.

1 Introduction

This paper describes certain aspects of the UCSD Tatami project, which has had the following main goals:

1. explore novel multimedia interface design principles, for easing the use of complex interactive systems;
2. build and use a generic distributed environment for cooperative work; and
3. verify distributed concurrent software.

We will discuss these goals in turn. The first is motivated by the difficulties that many practicing engineers have with formal methods tools. We have taken theorem provers as a typically difficult case, and have focused on finding ways to present proofs so that they are easier to understand and to construct.

The second goal is motivated by the observation that large projects have multiple workers, often at multiple sites with different schedules. It is therefore difficult to share information, coordinate tasks, and maintain consistency. We seek to ameliorate this with generic tools to support distributed cooperative work over the internet. An unusual feature needed by our application is to maintain the consistency and truth values of proofs parts in the face of constant change, distributed over both space and time, including insertion of new proof parts; truth values and logical dependencies are not well supported by standard tools for version and configuration management. Our tool uses fuzzy levels of credibility for components, rather than classical truth values. This tool is generic and has the potential for use in applications other than formal verification.

The third goal was chosen to introduce a realistic level of complexity in our proof tasks. Since this paper is mainly focussed on the first two goals, our discussion of the third is mainly limited to some generalities and some references to work where details can be found. On the other hand, we do discuss some specific user interface problems that arise in trying to provide better support for formal verification.

Formal methods have been used to prove the correctness of software systems, but this is known to be very difficult. Web technologies such as multimedia applets, Java, XML, XSL, and JSP offer opportunities to reduce this difficulty that have not yet been much explored. Since we wish to assist ordinary software engineers in using formal methods to design and verify complex systems, an important task for our project is to find better ways to

[1]This research was supported in part by National Science Foundation grant CCR-9901002, and by the CafeOBJ project of the Information Promotion Agency (IPA), Japan, as part of its Advanced Software Technology Program.

present and explain proofs. An examination of mathematics books and papers, even those in logic, shows that mathematicians almost never write formal proofs in the strict sense of mathematical logic. The only proofs written this way are very simple illustrations of formal proof methods, not proofs of genuine mathematical interest. This is because all but the simplest fully formal proofs are practically impossible to comprehend. Unfortunately, fully formal proofs are exactly what mechanical theorem provers produce.

To improve this sad situation, we suggest making explicit the motivation and structure of proofs, and integrating this information with relevant background and tutorial material. These recommendations are motivated by ideas from cognitive psychology, narratology and semiotics, as discussed further in Section 3. In particular, the structure of our proof websites was designed using *algebraic semiotics* [18, 19], which combines algebraic specification with social semiotics. Algebraic semiotics provides a way to formalize user interface designs and to compare them for quality, based on how well they represent what is most important in the underlying functionality; although it uses formal methods, algebraic semiotics does so in a way that remains sensitive to social context; see the discussion in Section 3.2.

Although full formal verification is an option, it is probably more practical exploit the task structure of formal methods without the burden of providing complete formal proofs for all steps; in this way, formality provides a discipline both for designing and using tools. See Section 2.3.3.

The Tatami system differs from other systems with which we are familiar in at least the following ways:

1. It automatically generates web-based interactive documentation.
2. It is designed to support distributed cooperative work, and has a distributed multi-project database and a specialized protocol to maintain database consistency, including the truth status of proof parts.
3. Many interface design decisions have been rigorously based on principles from cognitive psychology, narratology, semiotics, etc.
4. Specification is separated from validation, with a separate language for each.
5. A range of formality is supported, from full mechanical proof to informal arguments, using a fuzzy logic for the confidence values of assertions.
6. It heavily uses recent web and internet technologies.
7. It supports the design and validation of concurrent object oriented systems through its facilities for behavioral specification, based on hidden algebra.
8. The use of XML potentially supports to interchange of proofs with other theorem proving projects and systems.

The next section discusses the system architecture, while user interface design issues are discussed in Section 3, and conclusions and future research in Section 4. Two appendices give some sample code, and a third gives some formal definitions. This paper extends, updates and amalgamates work reported in [19, 22] and other papers. The latest information on the Tatami project can always be found at its URL, `www.cs.ucsd.edu/groups/tatami`.

2 Tatami System Design

This section sketches the Tatami system design, including:

1. its central component, the Kumo[2] proof assistant and website generator (see Section 2.2);
2. its databases (see Section 2.3);
3. its BOBJ specification language and underlying behavioral logic (briefly described in Section 2.4);
4. its Duck command language (see Section 2.5); and
5. its specialized communication protocol (see Section 2.6).

[2]This is a Japanese word for spider.

2.1 Architectural Overview and Some Implementation Details

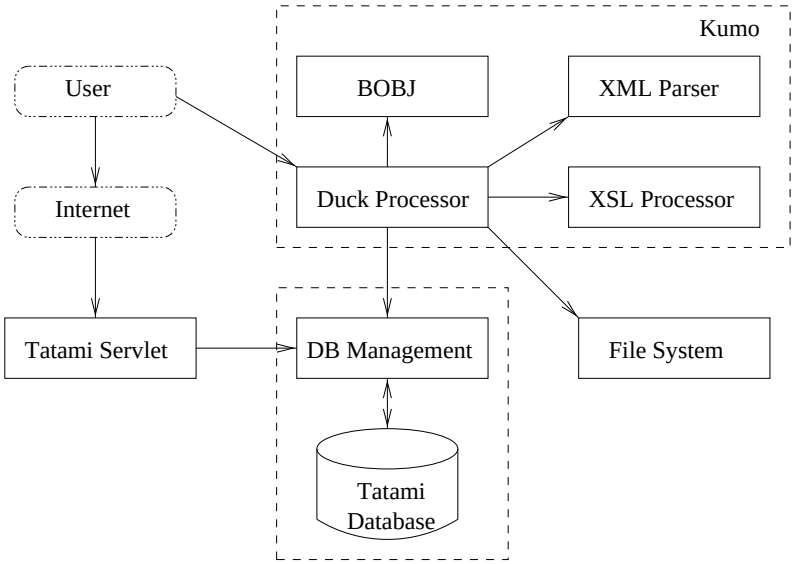

Figure 1. Tatami Implementation Overview

Figure 1 is an overview of the Tatami system architecture. Its most important components are the Kumo website generator and proof assistant, the tatami database, one or more proof engine (especially some version of OBJ), the tatami protocol, and a standard web browser. The Tatami servlets mentioned in Figure 1 are Java servlets, i.e., programs built using Java, cgi, and the HTML protocol, that implement the services of the tatami protocol (see Section 2.6.1). This architecture is modular, and in particular supports using a variety of logics. We do not want to re-implement the many complex algorithms that are provided by existing publicly available theorem proving systems. Instead, Kumo generates detailed proof scores that are sent to appropriate proof engines. Our current prototype uses the BOBJ (for Behavioral OBJ) [21] proof engine, but our Barista[3] proof server could also be used to wrap other proof engines, such as CafeOBJ [9] and OBJ3 [30]. The tatami database, BOBJ and Duck are implemented using Java technology, including JavaCC for parsing. XML files generated by Kumo are passed to a processor that uses an XSL style file to generate the HTML pages actually seen by users; users can write their own XSL style files for a different display style if they wish.

The following components are logic dependent and would have to be changed to support a different logic:

- a parser for the logic's syntax;
- a "provlet" for each inference rule, which is some Java code implementing that rule;
- an XSL style file containing an XML display "template" for each rule; and
- possibly a new proof engine and server.

Of course, these changes will in general require some non-trivial programming.

2.2 Using Kumo

Kumo executes proof commands which generate a proof tree, from which the system generates a website that documents that proof. This can be done in two modes, called *local* and *database*; cooperative work is only supported in database mode, since local mode places output in local files rather than the tatami database. Users

[3]This is an Italian word for a person who serves espresso.

mainly interact with Kumo through two specialized languages, BOBJ for (behavioral) specification, and Duck, which has commands for both proof execution and proof display. A typical session goes as follows (see also Figure 2):

1. Choose a project; then load a specification, or introduce a new specification.
2. Select an existing proof task, or introduce a new proof task.
3. Enter or edit a Duck program, and then execute it in local output mode; this produces or updates the corresponding proof website.
4. View the proof website on a browser.
5. Repeat from step 2.
6. When done, execute in database mode; then all subtasks of the selected task is entered into the tatami database, where they can be viewed by all project members.

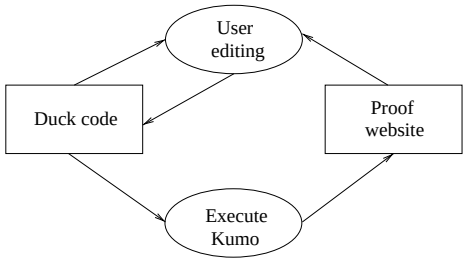

Figure 2. The Edit-Execute-Browse Cycle

While this kind of edit-execute-browse cycle (again see Figure 2) might seem old-fashioned to some, empirical studies [39] and our own experience have found that the current fad for direct manipulation interfaces for theorem proving is counter-productive for complex proofs, although it has value as a pedagogical aid for small proofs, e.g., for beginning students of formal logic. The main problem is that the data structures that underlie proofs of real interest are too large to navigate easily by direct manipulation, in part due to the difficulty of knowing where you are, because of the homogeneous structure of proofs. Another problem is that specifications evolve in real projects, and this should be effectively supported. Although these points are simple, they are important for the practical use of formal methods tools, and they are neglected by most current tools. This may be due to a Platonic bias on the part of their implementors, as well as many of their users. Such a bias tends to exclude complex issues arising from the social and cooperative nature of work, and to ignore the limited rationality and embodiment of real workers, not excluding mathematicians. Of course, this is not an argument against more sophisticated proof environments, e.g., parallel to the integrated proof environments (IDEs) used in programming, nor is it an argument against direct manipulation as such.

2.2.1 Proof Dependencies in Local Mode

Kumo maintains a log file of all proofs that are created in local mode. Information in this file is indexed first by project, and then by proof, where a project may contain several proofs, and each proof has the following:

1. The proof name and its goal;
2. A "digest[4]" of its specification;
3. Its truth value;
4. Its URL and a digest of its proof website; and
5. The names of all other proofs that use it.

[4]This is produced by a one-way fixed-length hash function.

Any proof in the same project can be used directly by giving its name. Before allowing this, Kumo checks the digest of its specification and of its proof website to ensure that the same specification is used, and that the proof website has not been edited manually. If the check succeeds, the existing proof and its status are used in generating the new proof, and the website of the existing proof is hyperlinked to the website of the new proof. If the check fails, then Kumo provides an error message. There are two ways to refer to a proof from a different project:

1. Give its full name, of the form `ProjectName.ProofName`; or
2. First declare to the project to be used, then give a short name, just as with proofs from the same project.

The second method is more convenient when many proofs are used from a single project that is different from the one in focus. Kumo does not allow a new proof to use the same name as an existing proof, unless the new proof has the same goal as the old one, or the old proof is not used in any other proofs. In the first case, the old proof is overwritten by the new proof, and the status of the new proof is propagated to all proofs that used the old proof.

2.3 Tatami Databases

Any practical implementation of formal methods must support bookkeeping for the complex dependencies among proof tasks, subtasks, and specifications, in the face of their mutual evolution, and the difficulties of coordinating work over a distributed group. The Tatami system uses a distributed database, consisting of a local database for each worker, which is conceptually divided into three components, which are described in the three subsections below. Coherence is maintained using a specialized protocol, which is described in Section 2.6.

2.3.1 The Project Database

The Project Database contains a list of projects, with their members and leaders. A project is started by placing it in this database, and whoever does so becomes its leader. Project Database information can only be modified by its leader.

2.3.2 The Specification Database

The Specification Database has two parts, one for data used for values, the other for abstract (software) machines. A number of relations may be defined[5] on these, including:

1. Importation of one specification by another.
2. Equivalence and enrichment of specifications.
3. Refinement for abstract machines.
4. Evolution from a previous version.

There are also some important relations that hold among these relations, for example that enrichment implies refinement, and that equivalence implies enrichment. Implementing these meta-relations would allow some further useful kinds of automation to be supported.

2.3.3 The Proof Database

The Proof Database keeps track of the support given for tasks and subtasks, which can range from fully formal mechanical proofs to informal “back of envelope” arguments. Alternative validations of the same task are also allowed. Instead of a precise description of how these capabilities are achieved (for which see [22]), we give an overview that stresses their importance for a system intended to be used in a practical way in real projects.

[5] At the time of this writing, items 2 and 3 were not yet implemented.

Recall that fully formal proofs are not found in real mathematics, for the very good reason that they would be too difficult to comprehend, and too difficult to produce. The same reasons apply to formal proofs in computer science. Commercial software projects usually cannot afford the overhead that would result, especially in view of the burden of updating proofs in the face of evolving requirements and the consequently evolving specifications and code. However, *informal* or *semi-formal* arguments for correctness are often developed, and it would surely be a good thing if these could be preserved and integrated into the documentation of the system as it is produced (see the discussion in [15]). Moreover, the overhead of formal proofs may be worthwhile for some key aspects of systems, especially "safety critical" systems, where human life is at risk. As for multiple proof attempts, these often arise when working on a difficult proof task, because it is unclear which attempt will succeed, if any.

It is interesting to notice that ordinary 2-valued logic is unsuitable for this situation, because the credibility of informal arguments in general lies somewhere between true and false; this suggests using a fuzzy logic for credibility. Because the usual fuzzy logic [53] does not capture the multiplicative effect of plausible argumentation, in particular, the fact that a chain of plausible steps becomes increasingly implausible as it gets longer [14], we use the fuzzy logic of [13], which represents conjunction with multiplication and disjunction with the maximum operation[6]. This logic allows for the cumulative effect of uncertainty in chains of reasoning, while taking the most promising choice when there is more than one alternative, that is,

$$\begin{aligned} [\![P \text{ and } Q]\!] &= [\![P]\!] * [\![Q]\!] \\ [\![P \text{ or } Q]\!] &= \max\{[\![P]\!], [\![Q]\!]\}\,, \end{aligned}$$

where $[\![P]\!]$ indicates the credibility of proposition P. For example, if we have two proof sketches for a goal G, one of which relies on informal proof steps P_1, P_2 having credibilities .7, .8, respectively, while the other relies on informal steps Q_1, Q_2, Q_3 with credibilities .9, .8, .9, respectively, then the credibility we infer for G is $\max\{.56, 648\} = .648$. Note that each chain may involve any number of correct applications of formal rules, because these all contribute a multiplicative factor of 1. See [54] for a promising alternative approach that is based on "computing with words."

The proof database can be seen as organized by a data structure called a **2-dimensional directed ordered acyclic graph**, or **2-doag** for short. Nodes in a 2-doag are labeled with validation tasks. Instead of edges, "fans" come out of nodes, where each fan represents a validation step for the task at its source (see Figure 3, noting that the leftmost and rightmost lines are just to make the picture look more like a fan; they do not lead to nodes). A fan can have zero or more nodes as its targets; these represent subtasks generated by the validation step. Furthermore, there can be any number of fans coming out of any given node, representing alternative ways to validate the task on that node.

Navigation through 2-doags is described by **2-occurrences**, which are sequences of pairs of positive integers, such as (1,2)(2,2)(1,1); given a node, such a sequence points to at most one other node, by taking the first number in each pair to indicate a fan and the second a target node of that fan. Thus the above sequence says: take the second target node of the first fan, from there take the second node of the second fan, and then take the first node of the first fan; this is illustrated in the left part of Figure 3. A formal description of 2-doags is given in Appendix C. These structures may be further specialized for our application as follows:

- A **truthdoag** is a 2-doag with a fuzzy truth value $t(n)$ at each node n. A boolean expression defining $t(n)$ is also associated with node n; it is the disjunction of the expressions for the fans going out from n. We use the fuzzy logic of [14] to evaluate the Boolean expressions; truth value 1 means there is a formal proof, while 0 means there is a formal disproof.
- A **proofdoag** is a truthdoag such that each node has a validation task; for formal proofs, each fan corresponds to some proof rule reducing the task at the root of the fan to the tasks on its leaves. The right part of Figure 3

[6]The usual fuzzy logic represents conjunction with minimum and disjunction with maximum [53]; the more general approach introduced in [13] uses the notion of "complete lattice ordered semi-group" (or "closg" for short), a structure that has more recently become popular under the name "quantile," and of which the logic of Kumo is the special case of the unit interval with multiplication.

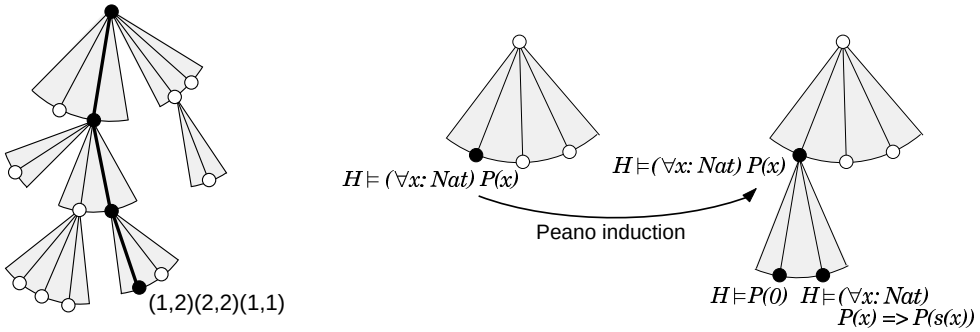

Figure 3. 2-occurrences and Peano induction

shows a proofdoag update that applies Peano induction for the natural numbers, to prove $(\forall x : Nat)P(x)$ from some hypotheses H.

2.4 Hidden Algebra and BOBJ

The Tatami project uses an approach to specification and verification called *hidden algebra*, which allows models that only satisfy their specifications *behaviorally*, in that they appear to exhibit the required behavior under all relevant experiments. This is important because many clever implementations used in practice only satisfy their specifications in this sense. Hidden algebra extends standard many sorted algebra by distinguishing between "visible" sorts used to model data, and "hidden" sorts used to model states. This framework provides simple and natural ways to define behavioral versions of basic concepts including equivalence of states, satisfaction of properties, and refinement of specifications. Standard equational deduction generalizes with some small changes, but more powerful inference rules are needed for proving most behavioral properties. The most important of these rules are versions of *coinduction*, and the project has been developing a series of increasingly powerful coinduction rules, in order to obtain proofs that are conceptually simple and yet highly mechanizable [44, 46, 47]. The most recent, called **circular coinductive rewriting**, is implemented in our BOBJ system in a fully automatic form [21, 46]. Recent research has shown that proving behavioral satisfaction is undecidable, in the sense that there can be no complete recursively enumerable set of inference rules [7]; therefore there can be no final resting point on our quest for simple but ever more powerful rules and algorithms.

The language of the BOBJ system [21] extends the classical algebraic specification language OBJ [30] by adding support for hidden algebra; in this it is similar to the CafeOBJ language [9]. The Tatami system and Kumo further extend the logic by allowing first order sentences with behavioral equations as atoms, plus induction for initially defined data types. Hidden algebra was originally proposed in [16], and has been elaborated in many subsequent publications, including [25, 26, 27, 28, 46, 47, 45], which should be consulted for details. Hidden algebra can handle all the main features of modern software systems, including states, classes, subclasses, attributes, methods, abstract data types, concurrency, distribution, nondeterminism, generic modules, and even logical variables (in the sense of logic programming), for which see [27].

This approach regards code as secondary, because it can be (relatively!) quickly written or even automatically generated from specifications that are sufficiently modular and detailed; moreover, empirical studies show that little of software cost comes from errors in coding [4]. This implies that a focus on specification and verification at the design level, avoiding the ugly complications of programming language semantics, will be the most productive.

Behavioral logic is a diverse research area containing many approaches, including the original hidden algebra of [16] and subsequent improvements in [20, 26, 25], the coherent hidden algebra of Diaconescu [10, 9], the observational logic of Bidoit and Hennicker [2, 3, 31], and a new generalization of hidden algebra that tries to treat all these variants in a uniform way [47, 28]. These approaches fall into two broad categories, depending on whether or not a fixed data algebra is assumed for all models. All proof rules in use are sound for all these

logics, but all of them are also incomplete [7]. Padawitz's "swinging types" are a powerful but less closely related approach [42].

2.5 The Duck Language

The Duck language includes commands that implement proof rules for hidden first order logic. These rules concern statements in the style of Gentzen's "natural deduction," which are "proof tasks" $H \vdash G$ with hypotheses H to the left of the "turnstile[7] symbol" $\vdash$, and with goals G to its right, where H and G are each sets[8] of sentences (represented as lists). Rules of inference are traditionally written as "fractions." For example,

$$\frac{H \vdash P,\ \ H \vdash Q}{H \vdash P \wedge Q}$$

is a rule for conjunction. Rules in such a form can be applied either backward or forward, which means bottom-up and top-down, respectively. For example, a backward application of the above rule reduces the task of proving a conjunction to that of proving its two constituents, while a forward application says that the conjunction is proved if its two antecedent proof tasks have been proved. When a logical operation appears as a result of a forward rule application, it is called an "introduction rule" for that operation; so the above rule is a conjunction introduction rule. Similarly, when a logical operation disappears when used in the forward direction, it is called an "elimination rule" for that operation.

A different distinction is whether rules change things to the right or to the left of the turnstile; these are called "right rules" and "left rules," respectively. Mechanical theorem provers generally apply their rules in the backward mode, and this is exactly what Kumo does. The right backward inferences are the most important, because they generate new subgoals from open subgoals in a way that (we hope) will eventually complete the proof. Left backward rules introduce new formulae into the set of hypotheses, in the hope that this will help with the proof. Two rules that can close an open subgoal are reduction (if it succeeds with value "`true`") and direct checking of whether the subgoal is in fact one of the hypotheses.

There is also a left version of the above conjunction elimination rule,

$$\frac{P, Q, H \vdash R}{P \wedge Q,\ H \vdash R}$$

and it also can be applied forward or backward. Right rules implemented in Kumo include the following:

- introduction rules for $\forall$, $\exists$, $\wedge$, $\vee$, and $\Rightarrow$;
- proof by contradiction;
- induction;
- (several kinds of) coinduction;
- reduction; and
- case analysis.

Kumo also implements some left rules, including lemma introduction, *modus ponens,* instantiation, congruence application, and conjunction introduction and existential quantifier elimination; the latter is better known as "Skolemization." The instantiation and congruence rules are special cases of equational deduction.

Some rules are "automatic" in the sense that they are always applied, unless explicitly inhibited by the user, while others must be explicitly invoked, because they require some user input; the automatic rules include reduction, and the right introduction rules for universal quantification, conjunction, disjunction and implication.

[7] Strictly speaking, the turnstile should have a subscript indicating the set of function and relation symbols currently in use.

[8] Actually, our implementation simplifies to the case where G is a single sentence. This is sufficient because a task $H \vdash G_1, ..., G_n$ can always be replaced by the n proof tasks $H \vdash G_1, \ldots, H \vdash G_n$.

Kumo generates a proof tree when it executes a proof described by some Duck code. For example, when the right conjunction rule is applied (backwardly) to a task $H \vdash A \wedge B$, then two new tasks, $H \vdash A$ and $H \vdash B$, are added to the tree. Each node of this tree, and each goal among its proof tasks, is given a unique system generated name, and may also have an optional user supplied name. The user-supplied names make it possible to apply proof rules in orderings that differ from the default.

After finishing a proof, Kumo translates its proof tree into XML for display. Duck has a distinct sublanguage for users to provide additional information for generating this XML, such as how to distribute proof steps over webpages[9]. Then Kumo uses an XSL file to generate HTML text for the proof tree; at this stage, Kumo adds explanations, links to tutorials, to machine proof scores, and to other proofs that are used (e.g. lemmas), etc.; this is partially illustrated in Appendix B. Different XSL files will generate displays in different formats.

2.6 The Tatami Protocol

A problem with using a distributed database is that local consistency can be lost if submitted proofs are inserted without being checked. For example, if a proof p_1 depends on a proof p_2, then the former should not be counted as verified unless p_2 has already been verified and entered. Inconsistencies can also arise when items are deleted. The tatami protocol maintains the consistency of the tatami databases, taking account of the following situations (see [29] for a detailed description):

1. If there are some items depending on an item *A*, then deletion of item *A* is disallowed.
2. If there are no items depending on item *A*, then the owner of item *A* can delete it.
3. Before using an item, existence of this item in the owner's database is checked.

Each message contains a 2-occurrence to specify the location of its update in the proofdoag. The occurrence of its parents in the sender's proofdoag are also sent, to enable detection of missing data. The message format is as follows:

1. project name;
2. a 2-occurrence for the update;
3. 2-occurrences of parents of the update; and
4. a tag indicating the kind of update, which may be submission, deletion, copy, validation, status update, etc.

The atomic entities sent to proof databases are fans. Each message contains exactly one fan together with its position in the proofdoag. Of course, workers sometimes want to submit more than one fan; this is accomplished by dividing the desired sub-doag into fans that are sent separately. Each fan $\mathcal{F}$ is assumed to have the following attributes:

1. a unique label which gives the position of $\mathcal{F}$ in the doag (occurrences can be used as labels, noting that one fan can have multiple occurrences); and
2. a message which encodes the fan $\mathcal{F}$ together with its label and all of its direct parents' labels (note that $\mathcal{F}$ can have zero, one or more parents) and the other data items discussed above.

We have used Kumo to formally prove the correctness of a version of the tatami protocol with respect to a communication medium that can lose or duplicate data [29], and we have also implemented it, as described below.

[9] If no information is supplied, then Kumo applies a set of default conventions which produce a reasonable output.

2.6.1 The Tatami Protocol Implementation

Kumo refines the tatami protocol into several services for communicating with the local servers at each site; these are implemented by servlets, built using Java, cgi and the HTTP protocol.

- **Project start service** This creates a new project with a given name; the user of this service becomes the leader of the new project. The leader's tatami database is updated, and other services as prepared, e.g., an empty waiting list is setup for subscription.
- **Subscription service** This tries to subscribe a new user to a project. When this service is requested, the user name is checked to ensure uniqueness in the project and its waiting list. If this succeeds, the requester is inserted in the waiting list, and remains there until the leader makes a decision. If the requester is accepted as a member, all project related information is copied from the leader's database to the requester's database, and the new member is added to all local databases.
- **Add service** This inserts new items into the databases of all project members, with the service requester as the owner of the new items. This service serializes the data dependence relation, so that the dependent data of an item is installed before the item itself is installed.
- **Deletion service** This tries to remove items from members' databases, noting that items only can be deleted by their owner. Also an item cannot be removed if there are other data that depend on it, or if someone else wants to use it. When the deletion conditions are satisfied, the deletion service blocks this item for future use, and then notifies all members' databases to remove it.
- **Unsubscription service** This allows a non-leader to leave a project. The project leader takes over all data owned by this member, and other users' databases ownership and membership information is updated.

3 User Interface Design Principles

User interface issues are important, because we want to help ordinary software engineers, who tend to be averse to formal methods, especially formal proofs. Cognitive science, semiotics (the theory of signs), narratology (the theory of stories) and even cinema, have influenced our design. Our most novel technique is algebraic semiotics, which provides systematic ways to evaluate the quality of user interfaces, including proof presentations. We hope that this research will contribute to improving the currently rather shaky scientific basis of user interface design.

3.1 Narratology

Finding a non-trivial proof usually requires exploring many dead ends, errors and misconceptions, some of which may be very subtle. Therefore the process of proving can be full of disappointed hopes, unexpected triumphs, repeated failures, and even fear and interpersonal conflict. All this is typically left out when proofs are written up, leaving only the map of a path that has been cleared through the jungle. We believe that proofs can be made much more interesting and understandable if some of the conflicts that motivate their difficult steps are integrated into their structure, instead of being ignored. As Aristotle said, "*Drama is conflict.*" What this means here is that restoring conflictual information will add dramatic interest, making it more interesting to read the proof. Of course, this must be done with care, and it should not be overdone, just as in a good novel or movie. We have used these ideas to structure some of our proof websites, since proof obstacles are exactly what is needed for creating drama.

An important resource for this is the theory of oral narratives developed by William Labov [33], who showed that these have a precise structure, which includes the following:

1. an optional *orientation section,* which provides basic orientation information, such as the time and place of the story;

2. a sequence of so-called *narrative clauses* which describe the events of the story;
3. the *narrative presupposition,* which by default assumes that the ordering of the narrative clauses corresponds to the temporal ordering of the events that they describe;
4. *evaluative material* interleaved with the narrative clauses, which "evaluates" the events, in the sense of relating them to socially shared values; and finally
5. an optional *closing* section, which may contains a "moral" or summary for the story.

The above follows [36, 37], which describe more recent developments than [33]. Aristotle [1] also gave some useful guidelines, including unity of time and place, having a beginning, middle and end, and the skillful use of language, especially metaphor, for which one can consult the work of Lakoff [35] and other cognitive linguists.

The work of Joseph Campbell and Christopher Vogler [52, 8] on the role of characters in stories, especially the role of heroes, is also relevant; the dramatic importance of having the hero tested by obstacles is emphasized by these authors. Some experimental results from prior work on multimedia instruction [23] are also relevant, suggesting that narrative, especially when it is oral (i.e., in the audio medium), is important for controlling the interpretation of material in other media. It is also well known in cinema that exactly the same scene with a different narrative can have a completely different interpretation and emotional effect; music can also have a powerful effect on interpretation. Our proof website design conventions are discussed in Section 3.3 draw on some of these ideas for their justification.

3.2 Algebraic Semiotics

A basic insight due to Ferdinand de Saussure [50] is that signs should not be considered in isolation, but rather as elements of *systems* of related signs, including their structural aspects. For computer scientists, it is natural to use tools from algebraic specification (e.g., see [24]) to formalize the intuitive notion of a *sign system*, as a loose algebraic theory (consisting of a signature and some axioms) plus some further structure specific to semiotics, including constructors, a hierarchy of levels for signs, and priorities among constructors [18]. A related insight is that *representations* can be seen as translations or maps from one sign system to another [18]; it is similarly natural to formalize these translations using the algebraic specification notion of *theory morphism.* Our case studies show that maps between sign systems in general do not fully preserve structure, and in particular, must involve *partial* functions. These considerations motivate the precise definition of *semiotic morphism* that is given in [18]; a key point is that the *quality* of a representation can be examined in terms of what is preserved by its semiotic morphism.

User interfaces are of course prime examples of representations, and it is natural to study them using semiotic morphisms which map from the sign system of the underlying functionality to a sign system for displays; the quality of the interface is then measured by the quality of its semiotic morphism. Details, some of which are quite technical, are omitted here, but may be found in [19] and [18]. In particular, note that this theory provides a natural way to handle multimedia displays.

3.3 Proofwebs and the Tatami Conventions

Professional advice for user interface design in general, and for website design in particular, nearly always calls for using style guidelines to produce a uniform "look and feel" that is appropriate for the particular application, e.g., see [43, 51]. We have developed the following *tatami conventions* (updated from [19, 22]) as style guidelines for the proof websites generated by Kumo. To clarify the discussion, we distinguish sign systems for *abstract proofwebs* and *display proofwebs*; the first contains the proof information, while the second also includes display information. We will use the word "*unit*" for any block of information of the same kind in a display proofweb.

1. *Homepages* are provided for every major proof part; these serve to introduce and motivate the problem to be solved and the approach taken to the solution.

2. *Tatami pages* are the basic constituents of display proofwebs; each tatami page has one or more *proof units* showing its inference rule applications, interleaved with one or more explanation units; it is feasible to have both on the same web page because there should only be a small number of proof steps per page (about 7 non-automatic rules works well). Tatami pages are also called "proof pages."
3. The *explanation units* of tatami pages are prover-supplied informal discussions of proof concepts, strategies, obstacles, etc.; these may contain graphics, applets, and of course text.
4. Tatami pages can be browsed in an order designed by the prover to be helpful and interesting to the reader; if possible, these pages should tell a story about how obstacles were overcome (or still remain); this order will be called the *narrative order*.
5. Major proof parts, including lemmas, have their own subwebsites, each with the same structure as the main proof, including homepage and explanation units. These appear in a separate dedicated persistent popup window.
6. Tatami pages also have associated formal proof scores, which appear in another separate popup window when summoned from a tatami page. It is convenient to have a separate window because users usually want to look at the proof and explanation at the same time as the proof score. Readers can also request proof score execution, and the result is displayed in the same window as the score, so that one can easily alternate between them. (The proof score is sent to an OBJ server and the result is returned for display.)
7. Major proof parts can have an optional closing page, to sum up important results and lessons; these appear in the same window as their tatami pages.
8. Formal proof steps are automatically linked to pre-existing tutorial background pages; e.g., each application of induction is linked to a webpage that explains the kind of induction used. Tutorial pages have their own dedicated persistent popup window.
9. A menu of open subgoals appears on each homepage, and error messages are placed on appropriate pages. In database mode, a summary of this information may be seen in the status window, which is a specialized popup that reports any currently open subgoals.

These conventions have the effect of integrating proofs with the information that is needed to motivate, understand and debug them; the intention is to make proofs easier to do and to understand, and to display information in a way that facilitates typical patterns of use.

3.4 Justifying the Design Guidelines

This subsection applies the techniques discussed in Sections 3.2 and 3.1 to justify some of our design guidelines for the proof websites generated by Kumo. We first justify the tatami conventions given in Section 3.3 using the same enumeration as in that section. This list of justifications mainly draws on narratology, while the justifications that are given after this will mainly use algebraic semiotics.

1. Having homepages for major proof parts is motivated by the orientation sections in Labov's theory of story structure [33]; homepages orient proof readers to what is to follow. Homepages appear in the same window as their tatami pages because they are part of the same narrative flow.
2. Interleaving prover-supplied informal explanations with proof steps was suggested by the similar interleaving of narrative and evaluative material in stories (see Section 3.1 and [33]); the evaluative material provides the motivation for important proof steps, by relating them to values shared among the appropriate community of provers. Limiting the number of non-automatic proof steps on tatami pages to approximately 7 is consistent with classic work of Miller on limitations of human cognitive capacity [40].
3. Explanation units correspond to the evaluative material in Labov's theory.
4. The idea of a giving a "narrative" order to tatami pages comes from the theory of stories [33]; the idea of including obstacles comes from Campbell [8] and others on "heroic" narratives.

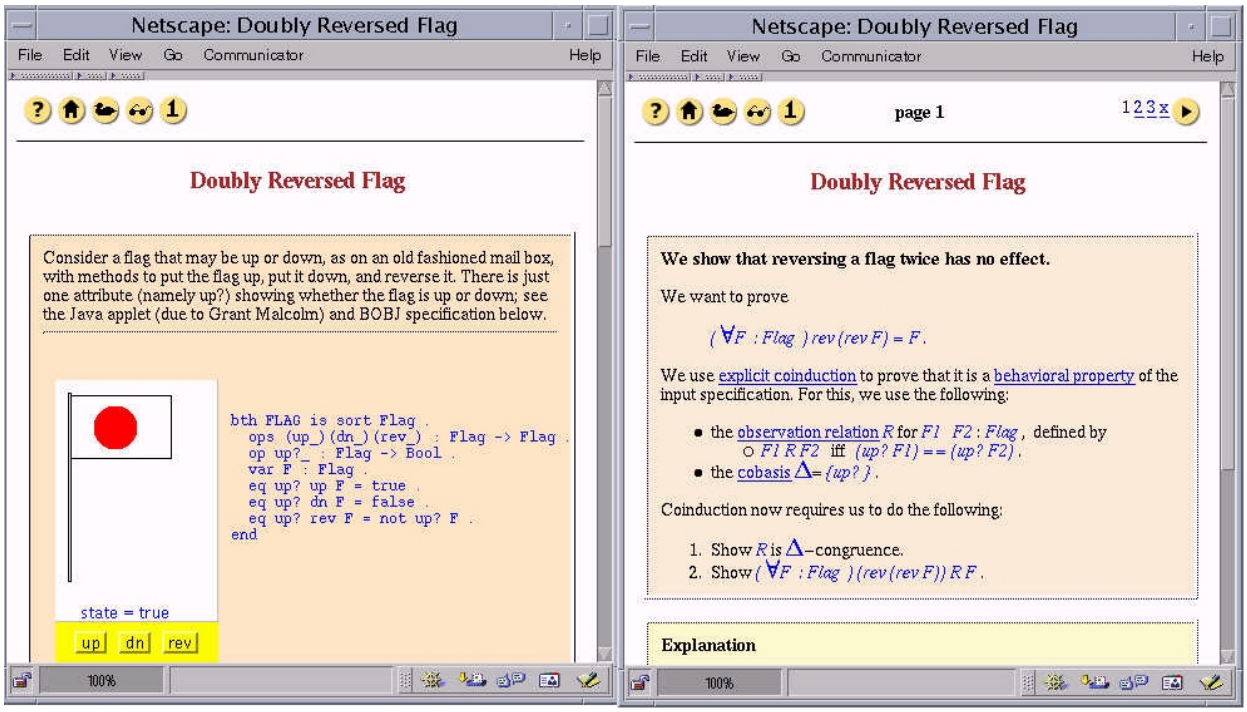

Figure 4. A Typical Tatami Homepage and Proof Page

5. Having separate hyperlinked websites for major proof parts is similar to the way that flashbacks and other temporal dislocations occur in stories. It is helpful to have them in a separate window in order to clarify their relation to the main sequence of proof steps.
6. Separating mechanical proof scores from the proof pages that generate them allows hiding the most routine details of proofs, just as human proofs often omit details in order to highlight the main ideas [38]; however, proof readers can still view them, and even execute them.
7. The optional closing webpages for proof websites are inspired by Labov's theory of story structure [33], and these also appear in the same window as proof pages, again because they are part of the same narrative flow.
8. Linking proof steps to tutorial pages can be motivated by connectionist theories which assert that concepts are organized as linked structures, e.g. [49]. (However, other research on concepts regards such theories as overly simple [48, 34].)
9. Open subgoals are important to provers when they read a proof, since proving new results is a major value within this community. The open subgoals are the leaf nodes of the current proof tree, and hence form a list in a natural way.

Now we give some justifications that use ideas from algebraic semiotics. The basis for these arguments is that any display to users can be seen as a semiotic morphism from the sign system for abstract proofwebs to that of display proofwebs, and that these morphisms can be compared for their quality, based on how well they preserve source structure and content; see [18] for more details (but note that[18] refers to an older version of Kumo).

- **Windows:** The main contents of a display proofweb are its proof steps, informal explanations, tutorials, and mechanical proof scores. These four are also the main contents of abstract proofwebs, and their preservation has much to do with the quality of their representation. These four basic sorts of the abstract data type for proofwebs are reflected in our choice of windows for displaying them. Because tatami pages are the main constituent of proofwebs, theirs is the master window, and because explanation pages are so closely linked,

they share that window; each unit is enclosed is its own "box." Tutorial and machine proof score pages each have a separate window.

- **Backgrounds:** Each major sort of unit has its own background color: proof units have light beige, explanations have light yellow, tutorials have yellow marble, and proof scores have light purple. Although the choice of colors is somewhat arbitrary, and is easily changed by editing the XSL file, their distinctness reflects the importance of distinguishing these four units.
- **Navigation:** Similar considerations hold for navigation. Each page has a title, supplied by the user in the Duck script. Buttons are used to move to other pages of the same sort, and to open widows that display information of other sorts. Each persistent window has somewhat different layout and navigation buttons, reflecting its different typical uses. For example, the master tatami window has "NEXT" and "PREV" buttons to step through the narrative ordering of tatami pages, as well as buttons that support conventional tree traversal ("LEFT", "RIGHT", etc.).
- **Mathematical Formulae:** We use `gif` files for mathematical symbols, in a distinctive blue color, because mathematical signs come from a domain that is quite distinct from that of natural language. (We hope to use MML when it becomes available.)

Some additional applications of semiotic morphisms to the user interface design of the Tatami system are described in [19], in a more precise style than here, although they are based on an older version of the system. For example, [19] shows that certain early designs for the status window were incorrect because the corresponding semiotic morphisms failed to preserve certain key constructors. The graduate user interface design course at UCSD uses algebraic semiotics (see `www.cs.ucsd.edu/users/goguen/courses/271`), and more information can be found there.

3.5 Dynamic Algebraic Semiotics

Classical semiotics is concerned with static signs; it does not allow for signs that change in response to user input, or that move on their own. This section sketches a new generalization of algebraic semiotics that can handle dynamic interfaces, by extending its foundation from classical algebra to hidden algebra. As a simple example, consider the problem of designing that part of the Kumo interface that supports browsing proofs. Kumo provides one set of buttons to traverse in the proof author's narrative order, labeled "NEXT" and "PREV", as well buttons "LEFT", "RIGHT", etc. for the usual tree oriented traversal. It is common practice to draw an automaton having one state for each proof tree node, and a transition label for each traversal button. But this does not allow for the fact that different proofs have different structures, and thus different automata, nor does it account for the different displays that are produced in each state, nor for the variety of possible implementations of transition lookup, e.g., using lists, arrays, or hash tables. An automaton can describe how a single proof instance can be navigated, but it cannot describe the general method which generates proof navigation support for any given proof, nor the way that this method is implemented, nor the quality of the resulting interface.

In fact, despite the formal character of the model itself, the construction and use of transition diagrams (or the corresponding automata) in user interface design is intuitive, and does not provide an adequate basis for a rigorous mathematical analysis of possible designs. In order to address the display, implementation and quality questions raised above, the automaton model must be supplemented in various *ad hoc* ways, whereas our approach can handle all of these within a single unified framework. Another example of dynamics in Kumo that would be difficult to handle with traditional user interface modeling techniques is the facility to execute the proof script for a proof part by downloading it to a BOBJ server and then viewing the result on a local browser as it executes.

This is not the place for details, but we can say that hidden algebra (see Section 2.4) provides a precise way to handle both the display and implementation aspects of examples like that described above, and the corresponding extension of semiotic morphisms gives a precise basis for comparing the quality of interface designs realizing the desired dynamics, without bias towards any particular implementation.

3.6 Gibsonian Affordances

An "affordance" in the sense of Gibson [11, 12] may be defined as "a capability for a specific kind of action, involving an animal and a part of its environment." For example, a cup provides an opportunity to exercise an affordance for drinking. Affordances are realized by the sensory-motor schemata (in the suggestive terminology of Piaget, though with a rather different meaning) that support some particular kind of action, such as drinking coffee from a cup. Instead of constructing, storing, and then using (massively large and complex) representations of objects (such as cups) from the external world, we rely upon the world to provide us with the information that we need, when we need it; this "offloading" of representation into the world can provide an enormous gain in efficiency, as shown in second generation AI research, e.g., that of Brooks [5, 6]. It seems promising to integrate a theory of affordances with algebraic semiotics, in the hope that this could produce new insights for each field, and advance the state of theoretical research in user interface design. One difficulty is that affordances as originally formulated are not well suited to symbolically mediated activities; however it seems that socially-based semiotics described in [17] could provide a good basis for resolving this difficulty; also work of Ittelson [32] and Norman [41] seems relevant (though [41] is a bit naive); the "image schema" notion that developed in modern cognitive linguistics [34, 35] is also very relevant.

4 Conclusions and Future Research

The Tatami project has developed in a perhaps surprising diversity of directions, including theoretical foundations of behavioral verification for distributed concurrent systems (using hidden algebra), web-based system development, and multimedia interface design (using algebraic semiotics). Although significant progress has been made, a great deal still remains to be done in each of these three areas. Fortunately, they are mutually reinforcing. For example, improvements in the user interface design of the Tatami system and its Kumo prover make it easier to do proofs in hidden algebra, which in turn inspire further developments in the theory, which in turn inspire further improvements to the system. We have also discovered how to handle dynamic displays, based on what amounts to a new kind of "dynamic" semiotics.

Some topics that it would be interesting to explore in future research include: making Kumo's proof displays even more interactive, for example, by using audio, avatars, chatrooms, and/or archetypal characters; extending our use of fuzzy logic with a fuzzy critical path algorithm that can choose the open subgoal having the greatest potential for increasing the overall credibility of a proof; extending the theory of semiotic morphisms to dynamic displays; and integrating Gibsonian affordances with algebraic semiotics (along the lines discussed in Section 3.6 above). In addition, we hope to use the Tatami system in teaching the UCSD graduate course on programming languages, which would stimulate further developments to the system, its interfaces, and its theory. Finally, we feel we are now in position to begin developing methodological guidelines for applying hidden algebra and the Tatami system to more challenging applications such as communication protocols.

Acknowledgements We especially thank Dr. Grigore Roşu for his extensive work on the Tatami project for his PhD thesis at UCSD. We also thank Prof. Kokichi Futatsugi for his encouragement and support through the CafeOBJ project, and we thank the international community interested in behavioral specification and verification for encouragement (see the behavior website, at the URL `www.cs.ucsd.edu/groups/tatami/behavior`).

References

[1] Aristotle. *Poetics.* Dover, 1997. Translation by S.H. Butcher; original from approximately 330 B.C.

[2] Gilles Bernot, Michael Bidoit, and Teodor Knapik. Observational specifications and the indistinguishability assumption. *Theoretical Computer Science*, 139(1-2):275–314, 1995. Submitted 1992.

[3] Michael Bidoit and Rolf Hennicker. Observer complete definitions are behaviourally coherent. In Kokichi Futatsugi, Joseph Goguen, and José Meseguer, editors, *OBJ/CafeOBJ/Maude at Formal Methods '99*, pages 83–94. Theta, 1999. Proceedings of a workshop held in Toulouse, France, 20th and 22nd September 1999.

[4] Barry Boehm. *Software Engineering Economics.* Prentice-Hall, 1981.

[5] Rodney Brooks. A robust layered control system for a mobile robot. *IEEE J. Robotics and Automation*, RA–2:14–23, 1986.

[6] Rodney Brooks. Intelligence without representation. *Artificial Intelligence*, 47:139–159, 1991.

[7] Samuel Buss and Grigore Roşu. Incompleteness of behavioral logics. In Horst Reichel, editor, *Proceedings, Coalgebraic Methods in Computer Science (CMCS'00)*, volume 33 of *Electronic Notes in Theoretical Computer Science*, pages 61–79. Elsevier Science, March 2000.

[8] Joseph Campbell. *The Hero with a Thousand Faces.* Princeton, 1973. Bollingen series.

[9] Răzvan Diaconescu and Kokichi Futatsugi. *CafeOBJ Report: The Language, Proof Techniques, and Methodologies for Object-Oriented Algebraic Specification.* World Scientific, 1998. AMAST Series in Computing, Volume 6.

[10] Răzvan Diaconescu and Kokichi Futatsugi. Behavioural coherence in object-oriented algebraic specification. *Journal of Universal Computer Science*, 6(1):74–96, 2000. Also technical report IS–RR–98–0017F, Japan Advanced Institute for Science and Technology, June 1998.

[11] James Gibson. The theory of affordances. In Robert Shaw and John Bransford, editors, *Perceiving, Acting and Knowing: Toward an Ecological Psychology*. Erlbaum, 1977.

[12] James Gibson. *An Ecological Approach to Visual Perception.* Houghton Mifflin, 1979.

[13] Joseph Goguen. L-fuzzy sets. *Journal of Mathematical Analysis and Applications*, 18(1):145–174, 1967.

[14] Joseph Goguen. The logic of inexact concepts. *Synthese*, 19:325–373, 1968–69.

[15] Joseph Goguen. Hyperprogramming: A formal approach to software environments. In *Proceedings, Symposium on Formal Approaches to Software Environment Technology*. Joint System Development Corporation, Tokyo, Japan, January 1990.

[16] Joseph Goguen. Types as theories. In George Michael Reed, Andrew William Roscoe, and Ralph F. Wachter, editors, *Topology and Category Theory in Computer Science*, pages 357–390. Oxford, 1991. Proceedings of a Conference held at Oxford, June 1989.

[17] Joseph Goguen. Towards a social, ethical theory of information. In Geoffrey Bowker, Leigh Star, William Turner, and Les Gasser, editors, *Social Science, Technical Systems and Cooperative Work: Beyond the Great Divide*, pages 27–56. Erlbaum, 1997.

[18] Joseph Goguen. An introduction to algebraic semiotics, with applications to user interface design. In Chrystopher Nehaniv, editor, *Computation for Metaphors, Analogy and Agents*, pages 242–291. Springer, 1999. Lecture Notes in Artificial Intelligence, Volume 1562.

[19] Joseph Goguen. Social and semiotic analyses for theorem prover user interface design. *Formal Aspects of Computing*, 11:272–301, 1999. Special issue on user interfaces for theorem provers.

[20] Joseph Goguen and Răzvan Diaconescu. Towards an algebraic semantics for the object paradigm. In Hartmut Ehrig and Fernando Orejas, editors, *Proceedings, Tenth Workshop on Abstract Data Types*, pages 1–29. Springer, 1994. Lecture Notes in Computer Science, Volume 785.

[21] Joseph Goguen, Kai Lin, and Grigore Roşu. Circular coinductive rewriting. In *Automated Software Engineering '00*, pages 123–131. IEEE, 2000. Proceedings of a workshop held in Grenoble, France.

[22] Joseph Goguen, Kai Lin, Grigore Roşu, Akira Mori, and Bogdan Warinschi. An overview of the Tatami project. In *Cafe: An Industrial-Strength Algebraic Formal Method*, pages 61–78. Elsevier, 2000.

[23] Joseph Goguen and Charlotte Linde. Optimal structures for multi-media instruction. Technical report, SRI International, 1984. To Office of Naval Research, Psychological Sciences Division.

[24] Joseph Goguen and Grant Malcolm. *Algebraic Semantics of Imperative Programs*. MIT Press, 1996.

[25] Joseph Goguen and Grant Malcolm. Hidden coinduction: Behavioral correctness proofs for objects. *Mathematical Structures in Computer Science*, 9(3):287–319, June 1999.

[26] Joseph Goguen and Grant Malcolm. A hidden agenda. *Theoretical Computer Science*, 245(1):55–101, August 2000. Also UCSD Dept. Computer Science & Eng. Technical Report CS97–538, May 1997.

[27] Joseph Goguen, Grant Malcolm, and Tom Kemp. A hidden Herbrand theorem: Combining the object, logic and functional paradigms. In Catuscia Palamidessi, Hugh Glaser, and Karl Meinke, editors, *Principles of Declarative Programming*, pages 445–462. Springer Lecture Notes in Computer Science, Volume 1490, 1998. Full version to appear in *Journal of Logic and Algebraic Programming*.

[28] Joseph Goguen and Grigore Roşu. Hiding more of hidden algebra. In Jeannette Wing, Jim Woodcock, and Jim Davies, editors, *FM'99 – Formal Methods*, pages 1704–1719. Springer, 1999. Lecture Notes in Computer Sciences, Volume 1709, Proceedings of World Congress on Formal Methods, Toulouse, France.

[29] Joseph Goguen and Grigore Roşu. A protocol for distributed cooperative work. In Gheorghe Stefanescu, editor, *Proceedings, FCT'99, Workshop on Distributed Systems (Iaşi, Romania)*, volume 28, pages 1–22. Elsevier, 1999. Electronic Lecture Notes in Theoretical Computer Science.

[30] Joseph Goguen, Timothy Winkler, José Meseguer, Kokichi Futatsugi, and Jean-Pierre Jouannaud. Introducing OBJ. In Joseph Goguen and Grant Malcolm, editors, *Software Engineering with OBJ: Algebraic Specification in Action*, pages 3–167. Kluwer, 2000. Also Technical Report SRI-CSL-88-9, August 1988, SRI International.

[31] Rolf Hennicker and Michel Bidoit. Observational logic. In *Algebraic Methodology and Software Technology (AMAST'98)*, volume 1548 of *Lecture Notes in Computer Science*, pages 263–277. Springer, 1999.

[32] William Ittelson. Visual perception of markings. *Psychonomic bulletin & Review*, 3:171–187, 1996.

[33] William Labov. The transformation of experience in narrative syntax. In *Language in the Inner City*, pages 354–396. University of Pennsylvania, 1972.

[34] George Lakoff. *Women, Fire and Other Dangerous Things: What categories reveal about the mind*. Chicago, 1987.

[35] George Lakoff and Mark Johnson. *Metaphors We Live By*. Chicago, 1980.

[36] Charlotte Linde. The organization of discourse. In Timothy Shopen and Joseph M. Williams, editors, *Style and Variables in English*, pages 84–114. Winthrop, 1981.

[37] Charlotte Linde. *Life Stories: the Creation of Coherence.* Oxford, 1993.

[38] Eric Livingston. *The Ethnomethodology of Mathematics.* Routledge & Kegan Paul, 1987.

[39] Nicholas Merriam and Michael Harrison. What is wrong with GUIs for theorem provers? In Yves Bartot, editor, *Proceedings, User Interfaces for Theorem Provers*, pages 67–74. INRIA, 1997. Sophia Antipolis, 1–2 September 1997.

[40] George A. Miller. The magic number seven, plus or minus two: Some limits on our capacity for processing information. *Psychological Science*, 63:81–97, 1956.

[41] Donald A. Norman. *The Design of Everyday Things.* Doubleday, 1988.

[42] Peter Padawitz. Swinging types = functions + relations + transition systems, 1999. Submitted to *Theoretical Computer Science.*

[43] Jenny Preece, Yovonne Rogers, et al. *Human-Computer Interaction.* Addison Wesley, 1994.

[44] Grigore Roşu. Behavioral coinductive rewriting. In Kokichi Futatsugi, Joseph Goguen, and José Meseguer, editors, *OBJ/CafeOBJ/Maude at Formal Methods '99*, pages 179–196. Theta (Bucharest), 1999. Proceedings of a workshop in Toulouse, 20 and 22 September 1999.

[45] Grigore Roşu. *Hidden Logic.* PhD thesis, University of California at San Diego, 2000.

[46] Grigore Roşu and Joseph Goguen. Circular coinduction. Technical Report CSE2000–0647, Dept. Computer Science & Engineering, Univ. California at San Diego, February 2000. Written October 1999.

[47] Grigore Roşu and Joseph Goguen. Hidden congruent deduction. In Ricardo Caferra and Gernot Salzer, editors, *Automated Deduction in Classical and Non-Classical Logics*, pages 252–267. Springer, 2000. Lecture Notes in Artificial Intelligence, Volume 1761; papers from a conference held in Vienna, November 1998.

[48] Eleanor Rosch. On the internal structure of perceptual and semantic categories. In T.M. Moore, editor, *Cognitive Development and the Acquisition of Language*. Academic, 1973.

[49] David E. Rumelhart and J.L. McClelland, editors. *Parallel Distributed Processing.* MIT, 1986.

[50] Ferdinand de Saussure. *Course in General Linguistics.* Duckworth, 1976. Translated by Roy Harris.

[51] Ben Shneiderman. *Designing the User Interface.* Addison Wesley, 1997. Second edition.

[52] Christopher Vogler. *The Writer's Journal: Mythic Structure for Storytellers & Screenwriters*. Michael Wiese Productions, 1992.

[53] Lotfi Zadeh. Fuzzy sets. *Information and Control*, 8:338–353, 1965.

[54] Lotfi Zadeh. Fuzzy logic = computing with words. *IEEE Transactions on Fuzzy Systems*, 2:103–111, 1996.

A Sample Duck Code

A Duck text for proving correctness of the traditional array-with-pointer implementation of stack is shown below. It has two parts, the first for proofs and the second for display. In addition to the main theorem, there is a key lemma, which is proved by induction, whereas the main goal is proved by coinduction. Each proof part begins with the keyword "`proof:`" and ends with "`[]`", which signals that Kumo should try to finish the proof using only its automatic rules (see Section 2.5). A name may be optionally given after the keyword "`proof:`". Comment lines begin with "`***`". More detail on this proof can be obtained from its homepage; see the URL `www.cs.ucsd.edu/groups/tatami/kums/exs/`; more information on the latest version of Kumo can be obtained at `www.cs.ucsd.edu/groups/tatami/kumo/`.

```
*** file: /net/cs/htdocs/users/goguen/kumo/stack/stack.duck
****************************************
style: beijing
****************************************
spec: /net/cs/htdocs/users/goguen/kumo/stack/stack.bob
****************************************
cobasis:
   op top _ : Stack -> Nat .
   op pop _ : Stack -> Stack .
[]
****************************************
proof: <<StackThm1>>
  goal:  pop empty = empty  and
   ((forall N I : Nat  A : Arr)  top push(N, <I, A>) = N  ) and
   ((forall N I : Nat  A : Arr)  pop push(N, <I, A>) = <I, A> ) .
[]
****************************************
proof: <<ArrayLemma>>
  goal: (forall I J N : Nat A : Arr)
   (I <= J  implies <I, put(N, J, A)> = <I, A>) .
  by: induction on I with scheme {0, s};
[]
****************************************
proof: <<StackThm2>>
  goal:  pop empty = empty  and
   ((forall N I : Nat  A : Arr)  top push(N, <I, A>) = N  ) and
   ((forall N I : Nat  A : Arr)  pop push(N, <I, A>) = <I, A> ) .
[]

****************************************
display: <<ArrayLemma>>
   title:  "A Behavioral Property of Pointer-Array Pairs"
   dir: /net/cs/htdocs/groups/tatami/kumo/exs/stack/lemma
   homepage: /net/cs/htdocs/users/goguen/kumo/stack/stacklemma.hp
   specexpl: /net/cs/htdocs/users/goguen/kumo/stack/stackspec.exp

 <<ArrayLemma>>
  expl: /net/cs/htdocs/users/goguen/kumo/stack/stacklemma.exp
[]
****************************************
display: <<StackThm1 StackThm2>>
   title:  "Implementing Stack with Pointer and Array"
   dir: /net/cs/htdocs/groups/tatami/kumo/exs/stack
   homepage: /net/cs/htdocs/users/goguen/kumo/stack/stack.hp
   specexpl: /net/cs/htdocs/users/goguen/kumo/stack/stackspec.exp
```

```
    closing:  /net/cs/htdocs/users/goguen/kumo/stack/stack.close

  <<StackThm1>>
   subtitle:  "This proof does not succeed, because a lemma is needed."
   expl: /net/cs/htdocs/users/goguen/kumo/stack/stack.exp1

  <<StackThm2>>
   subtitle:  "This time we succeed using the lemma."
   expl: /net/cs/htdocs/users/goguen/kumo/stack/stack.exp2
 []
```

B Sample XSL Code

The XML files produced by Kumo described in Section 2.5 are used together with an XSL style file to generate the HTML that is actually displayed by the user's browser. Below is the XSL code for the output of a conjunction elimination rule application:

```
<xsl:template match="Conjunction-elimination">
  <a href="{constant(conjel)}" target="back">
     Conjunction elimination </a>
  yields the following
    <xsl:apply-templates match="count"/>
  subgoals:
</xsl:template>
```

This says that the generated HTML will contain the text "*Conjunction elimination yields the following K subgoals:*" where the integer K is number of subgoals, obtained by calling another XSL rule, named "`count`"; the link to the conjunction elimination tutorial page, indicated by the underline, is also inserted, with URL named by the constant "`conjel`". (The list of subgoals is already present in the XML file being processed and therefore need not be inserted by this rule.)

C A Formal Description of 2-Doags

Here we briefly define the data structure that is used for organizing validations. If N is a set, then N^* denotes the set of finite lists over N. A **2-doag** D consists of a set N of **nodes**, a set F of **fans**, two functions $d_0 : F \to N$ and $d_1 : F \to N^*$ (called **source** and **target**), a set W of "labels" plus a labeling function $l : N \to W$, such that

1. for each node n, the fans with source n are ordered;
2. for each fan f, the target nodes of f are ordered;
3. let $D^\flat$ be the ordinary directed ordered graph constructed from D to have the same nodes N as D, and to have as its edges from n to n' pairs (f, n') such that n' is a target node of a fan f with source n, with edges ordered by the ordering of fans plus that of edges within fans; then $D^\flat$ must be acyclic.

The labeling function is just to accommodate the additional information attached to nodes mentioned in Section 2.3.3. Note that $D^\flat$ need not have a unique root and need not be connected. 2-doags are really a kind of hypergraph, and fans are really hyperedges, but we use the "2-dimensional" language because of its suggestiveness for our application. Similarly, a 2-occurrence in D is just a path in $D^\flat$.