Distributed Cooperative Formal Methods Tools*

Joseph Goguen, Kai Lin, Akira Mori, Grigore Roşu, and Akiyoshi Sato
Dept. Computer Science & Engineering, University of California at San Diego
9500 Gilman Drive, La Jolla, CA 92093-0114, USA
{goguen, klin, amori, grosu, asato}@cs.ucsd.edu

Abstract: *This paper describes some tools to support formal methods, and conversely some formal methods for developing such tools. We focus on distributed cooperative proving over the web. Our tools include a proof editor/assistant, servers for remote proof execution, a distributed truth protocol, an editor generator, and a new method for interface design called* **algebraic semiotics***, which combines semiotics with algebraic specification. Some examples are given.*

1. Introduction

Formal methods have been used to prove correctness of software, but this task is known to be difficult, for various reasons [5, 13]. New technologies – the internet, the web, multimedia, applets, etc. – offer exciting opportunities to reduce this difficulty that have not yet been much explored. We seek to build industrial strength formal method tools for distributed work by ordinary software engineers over the web, especially for concurrent systems. This involves several challenging subtasks, including a practical logic for concurrent systems, effective support for distributed specification and proving, and tools to help maintain complex websites. Algebraic semiotics helps with tool automation and provides a scientific basis for good interface design.

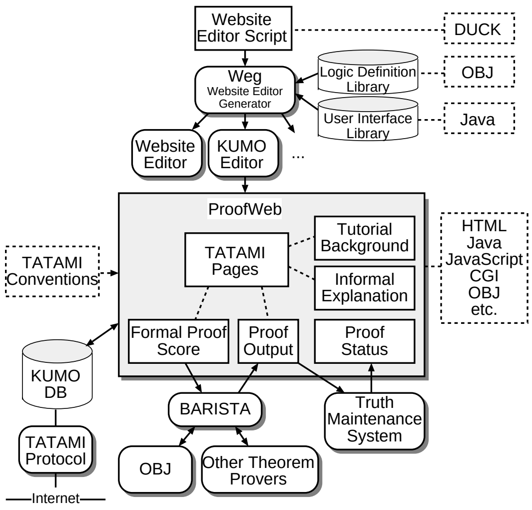

Figure 1: System Overview

Our system is called "TATAMI," after the natural fiber mats used in traditional Japanese homes. The main tool is KUMO (after a Japanese word for spider), a proof assistant and website editor. It links to the internet through the tatami protocol, which coordinates multiple workers on multiple proof versions. BARISTA executes KUMO proofs on remote servers. Any standard web browser can be used, since all interaction is implemented in Java and JavaScript. Some examples are at `http://lex.ucsd.edu/links`. We intend future versions of KUMO will be built by an editor generator. See Figure 1.

1.1. Tatami Conventions and ProofWebs

Our long term goal is to make the power of formal methods available to ordinary software engineers. Thus a major subgoal is to develop better presentations for proofs than in traditional mathematics, by integrating background and tutorial material, by improved proof structuring techniques, and by making motivation more transparent. This requires developing new data structures for proofs and their presentations, as well as input from areas like cognitive psychology and semiotics (which is the theory of signs). Our proof websites are structured by the following **tatami conventions**:

1. The main **tatami pages** describe proof steps.
2. Tatami pages can be browsed in a "narrative" order, designed to be helpful and interesting.
3. A user-supplied informal explanation page is linked to each tatami page, discussing proof strategies, obstacles, etc.
4. Major proof parts, including major lemmas, have their own homepage.
5. Proof homepages and explanation pages can have graphics, applets, and of course text.
6. Formal proof steps are hotlinked to tutorial background pages; e.g., each application of induction is hotlinked to its explanation webpage.
7. Each major proof part can have a "closing" webpage to sum up its lessons.
8. A formal proof score is generated for each tatami page; workers can execute these on remote servers, with results returned to the local site.
9. A frame with navigation buttons is provided for each kind of webpage.
10. Different kinds of webpage have different backgrounds.

These conventions are implemented in the **ProofWeb** data structure used by KUMO, and are describable in our DUCK language; they integrate proofs with information needed to understand them; an partial ADT is given in Appendix A. Figure 6 shows the structure of a typical ProofWeb, and Figure 2 shows screendumps of some typical tatami pages. Interactive Java applets can illustrate basic intuitions for the

*The research reported in this paper has been supported in part by the CafeOBJ Project of the Information Promotion Agency (IPA), Japan, on algebraic formal methods for software engineering.

problem and for key proof steps, allowing users to explore issues for themselves. This is important because reading specifications and proofs written by others can be painful. The ProofWebs we have built demonstrate a convergence of web capabilities with ways to make proofs more interesting and understandable.

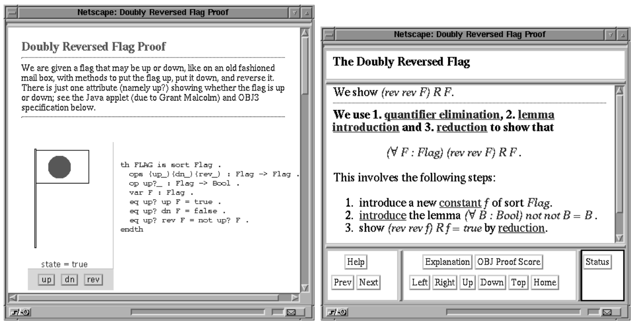

Figure 2: Screendumps of Typical Tatami Pages

Tatami pages appear in a fixed master window. Its navigation buttons include UP, DOWN, LEFT, RIGHT, plus PREV and NEXT; the latter two follow the narrative order. Proof status can be: `proved`, `unknown`, or `untrue`; this is displayed in a **status window** popup, which is updated locally by propagating changes up the proof tree. The status window also supports proof navigation. The third part of Figure 5 is a screendump of our current status window.

There is of course a huge literature on automating software engineering, and another on tools for formal methods, but we know of no prior work focused on distributed cooperative proving, or on proof representations to aid comprehension. This paper is far too short to review the literatures on cooperative work or on theorem proving, although these are very relevant. So we have only given citations that really influenced our work.

2. Distributed Cooperative Proving

Nearly all real software projects are distributed and cooperative, involving multiple workers, sites and subtasks; any practical implementation of formal must take account of this fact. Distributed cooperative proving requires careful bookkeeping for proof tasks, subtasks, dependencies, and specifications. Consistency is maintained by the tatami protocol, which updates the information in each local KUMO database. 2-doags and ProofDoags are abstractions of ProofWeb structure that help explain the operation and correctness of the tatami protocol.

2.1. Projects

This subsection motivates our approach to distributed cooperative work. A **project** is a set of linked websites devoted to closely related tasks. Although our design is decentralized, it needs an administrative site to register users, issue passwords, and host a community bulletin board; registration can be done using HTML query forms. A projects starts by placing a URL for its description in the community bulletin board; one worker is identified as the project leader. The leader also maintains the list of workers, which is used in broadcasting project messages.

Any worker can post a new proof task, pick up a proof task, and submit a partial proof. Workers can have local "ghost" proofs that cannot be seen remotely. Workers can delete their own proofs, but if some deleted subproof is used by another worker, its ownership is reassigned to whoever used it first. Workers can copy proofs of others; then the source files are copied as "ghost" proofs. Communication daemons monitor messages and initiate updates. When an atomic goal on a leaf node is reached, a proof score is created (by tracing the ProofDoag upward) and checked by a proof server, after which proof status is updated.

Work in sociology of mathematics has emphasized that proofs are constituted as such by communities of mathematicians [12]. It would be interesting to know in great detail how proofs function in and as communication, since we could use such information to improve our system.

2.2. 2-dimensional Doags

Alternative proofs of the same goal are supported by a data structure called a **2-dimensional directed ordered acyclic graph**, or **2-doag**. Acyclic graph rather than tree structure arises because results can be produced and used more than once; see also Shand's hypergraph proof representation [15]. The goal of these formal definitions is to get the right abstraction level for the operation and correctness of the tatami protocol.

A **2-doag** consists of a set N of **nodes**, a set E of **edges**, two functions $d_0, d_1 \colon E \to N$ (called **source** and **target**), a function $f \colon N \to F^*$ where $F \subseteq E^*$ is called the set of **fans**, plus **coloring functions** $c \colon N \to C$ and $c \colon E \to C$, where C is a set whose elements are called **users** or **workers**, such that

1. $E = \bigcup_{n \in N} Set(f(n))$, where $Set(f)$ denotes the set of elements that occur in a list f of lists;
2. $(\forall n \in N)$ each v in $f(n)$ has the same color;
3. $(\forall n \in N)$ each v in $f(n)$ has the same source; and
4. the ordinary directed ordered graph (N, E, d_0, d_1) is acyclic, where its edge ordering is given by the ordering of fans, plus that of edges within fans.

Note that (N, E, d_0, d_1) need not have a unique root or be connected. A **2-occurrence** is a sequence 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 an edge in that fan. Thus the above sequence says take the second edge of the first fan, then from the node you are now at, take the second edge of the second fan, then take the first edge of the first fan, as shown in Figure 3 (note that the leftmost and rightmost lines are just to make the picture look more like a fan; they are not real edges).

A **TruthDoag** is a 2-doag with at each node n a truth value $t(n)$ (one of `proved`, `unknown`, `untrue`) plus a boolean expression that defines $t(n)$ from the truth values of the leaf nodes of the fans out of n. The boolean expression at a node takes the disjunction of the boolean expressions of its fans. (The boolean expressions for most proof rules take the conjunction of their leaf nodes, but the rules for disjunction elimination and contradiction are exceptions.) This extends in a natural way to "fuzzy" truth values (in the unit interval, using maximum for alternative proofs, and multiplication for proof chains, as in [3]), to take account of informal and semi-formal subproof; clearly this is more realistic for practical software development.

A **ProofDoag** is a TruthDoag with the additional structure that each node has a proof task, and each fan corresponds to some proof rule reducing the task at the root of the fan to the tasks on its leaves.

We need some of these formal notions for stating and proving the tatami protocol, and others for our approach to interface design. The local KUMO database stores the ProofDoag information, plus a timestamp for each fan and URLs for the associated tatami, explanation, and proof score pages. Then one part of our correctness criterion for the tatami protocol is that everything it does to the database must leave invariant the condition: $t(n) = \texttt{proved}$ iff there is a proof for the proof task at node n. Figure 4 shows a ProofDoag update arising from application of Peano induction for the natural numbers. Another part of the KUMO database at each site stores basic information from all currently active projects, including a copy of each homepage and its URL.

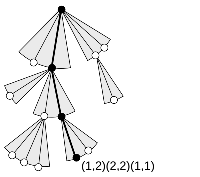

Figure 3: 2-Occurrence in a 2-Doag

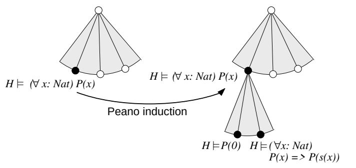

Figure 4: Peano Induction and ProofDoag

2.3. The Tatami Protocol

We focus on the logical consistency of each worker's information. Because the order of receiving messages over the net is unknown, and messages can even be lost, we must guarantee consistency, in that each result in each local database is always correct in the larger context, and so can be submitted without polluting anyone else's work. This involves handling the following situations:

- When deleting a fan: decide who will take it over, and wait for acknowledgment from all workers involved.
- When something is missing from a received message: save the message and wait for the missing part.

A unique label is attached to each message as a key for database search and as a consistent way to detect missing messages. The (unique) labels of its parents in the sender's ProofDoag are also attached, to detect missing data and assure consistency between the receiver and sender databases. Thus the message format is as follows:

- the project name and URL of its homepage;
- a unique label, obtained by concatenating the user's URL with a local counter;
- the unique labels of its parents in the sender's ProofDoag;
- the content of the message (e.g., a proof task, specification, fan, or a compound inference rule);
- a tag indicating the type of update: submission, deletion, copy, or proof status update.

To simplify exposition, we assume the atomic entities of messages are fans Of course, workers sometimes want to submit more than one fan; this is no problem, because any sub-ProofDoag can be divided into fans sent separately.

The following takes the viewpoint of an arbitrary but fixed worker. Each fan $\mathcal{F}$ has its label denoted by $l[\mathcal{F}]$. For each worker u and fans $\mathcal{F}_1, \mathcal{F}_2$, we say $\mathcal{F}_1$ **precedes** $\mathcal{F}_2$, denoted $\mathcal{F}_1 \prec_u \mathcal{F}_2$, iff $\mathcal{F}_1$ is a direct parent of $\mathcal{F}_2$. A **dependency graph** for a worker u is a directed graph having as nodes labels of fans, with an edge from $l[\mathcal{F}_1]$ to $l[\mathcal{F}_2]$ iff $\mathcal{F}_1 \prec_u \mathcal{F}_2$. If $\mathcal{F}_1 \prec_u \mathcal{F}_2$, then $\mathcal{F}_2$ cannot be sent to u until receipt of $\mathcal{F}_1$ is acknowledged. This prevents $\mathcal{F}_2$ from being a ghost. Dependency graphs keep track of unacknowledged messages between our worker and other workers in the same project. They are empty when communication succeeds. A dependency graph is needed for each other worker in the project. Each dependency graph is a standard dag; let $G[u]$ denote the dependency graph associated to worker u.

We assume the following are available: a function $gen\_message(\mathcal{F})$ that generates the message $l[\mathcal{F}]$ encoding the fan $\mathcal{F}$; functions $insert(l[\mathcal{F}], G[u])$ and $remove(l[\mathcal{F}], G[u])$ that modifies $G[u]$ by adding and removing, respectively, the node $l[\mathcal{F}]$ and appropriate dependency edges; a function $topological\_minimal(G[u])$ that gives a node in $G[u]$ whose associated fan is not preceded by any other (label of a) fan in $G[u]$; a daemon $send(u, m(\mathcal{F}))$ that sends the message $m(\mathcal{F})$ to worker u; a daemon $ack()$ which waits until an acknowledgment is received and then returns the acknowledgment pair $(u, l[\mathcal{F}])$; a daemon $receive()$ which waits until a message is received from some worker u and then returns a pair $(u, m[\mathcal{F}])$.

The main operation is SUBMIT, which sends a fan to all other workers in the same project. Informally, SUBMIT($\mathcal{F}$) first generates the encoded message $m[\mathcal{F}]$, then saves $m[\mathcal{F}]$, and appropriately modifies the dependency graphs of other workers (using $insert(l[\mathcal{F}], G[u])$). Each label has a $counter$ that records how many acknowledgments are outstanding for the message associated to that label. SUBMIT($\mathcal{F}$) sets $counter[l[\mathcal{F}]]$ to the number of workers in the project.

Sending, receiving and acknowledging messages is done using daemon processes. The first, SENDER_DAEMON, checks dependency graphs and sends appropriate messages (given by the function $topological\_minimal(G[u])$) to users u with $G[u] \neq \emptyset$. The second, RECEIVER_DAEMON, must take into account the fact that a worker can submit a fan $\mathcal{F}$, having as parent a fan $\mathcal{F}_2$ recently submitted by another worker, while a third worker did not receive $\mathcal{F}_2$ but did receive $\mathcal{F}_1$. In such a case, $\mathcal{F}_1$ could make no sense for the third worker, who should therefore save it and wait for $\mathcal{F}_2$. Thus RECEIVER_DAEMON first saves the received fan into a temporary (ghost) database, and then recurently delivers only the messages having all their parents available in the project database. The third, ACK_DAEMON, waits (using

$ack()$) for confirmations from other workers, and when a confirmation $(u, l[\mathcal{F}])$ is received, it removes the label $l[\mathcal{F}]$ from u's dependency graph, decreasing the counter of $l[\mathcal{F}]$, erasing the saved message when the counter reaches zero.

SENDER_DAEMON() guarantees that messages are sent in a consistent order to every worker, i.e., each worker receives dependent fans in the order they were submitted. There is no restriction on the order of sending independent fans, because they do not affect each other. This strategy guarantees that for each time and each local database, every `proved` result actually has a proof, which was our correctness criterion. Moreover, it guarantees that each fan in the database has a path from a root to that fan (nodes that state lemmas are considered roots); then every fan in the database occurs in the ProofDoag and thus can be accessed in a way consistent with that of the fan's sender.

RECEIVER_DAEMON and SENDER_DAEMON assure fairness, that each user automatically gets all the information in the project, even if his communication was disrupted for a time. The same algorithm works for deleting or updating database information; we just have to encode the desired information in a message in the same way. This protocol can be implemented using a standard internet protocol, e.g., IP.

2.4. BARISTA

Any proof, local or remote, can be checked by pressing the EXECUTE button on a proof score page; then a (Java) client applet is downloaded from the server (for security, the client must come from the server through a socket); the proof score URL is passed to the server; the server fetches the proof score, executes it, and sends the result back to the client applet. An experimental OBJ3 BARISTA (this is an Italian word for a person who serves espresso) server is now running at UCSD, and servers will be added for other provers and other sites later. OBJ servers prove atomic tasks by reduction; other servers may use other techniques.

2.5. Hidden Logic

We very briefly sketch the hidden (order sorted first order) predicate logic with equality (both ordinary and behavioral) used in our project. A **proof task** has the form $H \models_{\Sigma} P$, where H is a specification and P is a **proof goal**. Proof rules transform such tasks into atomic tasks that can be checked on a proof server.

Hidden algebra supports correctness proofs that are (relatively!) simple and mechanizable, and handles classes, subclasses, attributes, methods, local state, concurrency, distribution, nondeterminism, logical variables, abstract data types, generic modules, and more generally parameterized programming. Hidden algebra generalizes the process algebra and transition system approaches to non-monadic operations, so that it can take advantage of equations involving non-monadic methods and attributes. Hidden logic uses behavioral abstraction, or more precisely, behavioral satisfaction: hidden specifications characterize how systems *behave*, not how they are implemented; they provide a notion of behavioral type. **Coinduction** is new proof technique that is very effective for proving behavioral properties, e.g., behavioral refinement. See [9] for details.

3. Website Editing

Maintenance, the process of modifying software to meet user expectations in the face of evolving platforms, working conditions, business environments, etc., consumes far more effort than initial construction; the effects of evolution on formal methods are similar, but even worse [5, 13]. Websites with complex interaction (buttons, popups, etc.) and complex links are difficult to build and even more difficult to maintain; formal methods websites will thus operate under a severe double handicap. A website editor can greatly reduce this effort. *Website* editing differs greatly from *webpage* editing as supported by commercial products like MicroSoft's FrontPage, by addressing cross-page issues like link maintenance and site-wide design conventions; in fact, page editors can make such problems *more difficult*.

3.1. KUMO

KUMO is a proof assistant and website editor for our hidden logic; it produces ProofWebs, as described in Section 1.1, and operates in both batch and interactive modes. Its interface has parts for proof commands, ProofDoag display, and for other information. When used in batch mode, users write proof commands in a file which KUMO executes to the extent it can, putting the generated ProofWeb in the local database, and outputting error messages. ProofWebs generated in batch mode can be more compact than those in manual mode, because several proof steps may appear on one page; this also makes the result more readable; moreover, maintenance is easier in batch mode, since one can just edit the command file. In either mode, the following must be supplied:

1. A working directory, goal and spec, which is usually in a file that can be edited and reused.
2. Subgoals and proof rules, from which KUMO either generates new subgoal(s) or else proves the subgoal and modifies the proof status; in either case, ProofWeb pages are constructed and/or updated.

Users can select subgoals from the ProofDoag window by mouse click. Auxiliary information (e.g., explanations) can be provided directly or by reference to a file. The following describes some of the proof rules:

Induction is very general; an induction variable, its sort, and possibly some new operations and equations should be supplied [7]. For example, applying Peano induction to the proof task `H |=[S] (all x) plus(sum(x),sum(x)) = times(x,s(x))` yields the update shown in Figure 4.

Lemma Introduction for any subgoal, by giving a URL, which causes the lemma to be ftped, its status checked, and its subwebsite connected to the current subgoal. If proved, it is added to the premises of the current subgoal; otherwise, its proof becomes a new subgoal. Local lemmas (used only in the current proof) are given a subdirectory.

Automatic Rules can be applied without any user supplied information. For example, conjunction elimination is automatic; applied to $H \models A \wedge B$, KUMO updates the ProofDoag and tatami pages with new subtasks $H \models A$ and $H \models B$. The is also a `prove` command that simplifies a proof task and its descendants, applies reduction to equational subgoals, and returns all unproved subgoals.

Reduction sends equations to OBJ, and receives the result.

3.2. DUCK and WEG

Not only website content, but also website design evolves, though more slowly; for example, our tatami conventions have evolved greatly over the course of the project. To address this problem, we are building a tool to generate KUMO-like systems from specifications.

WEG generates an editor from specs for website structure (described as a sign system – see Section 4.1), a logic (given as an ADT in OBJ), and user interaction (described in the agent language DUCK). OBJ can be used for parsing, inference, and checking ground equations. DUCK code describes the interface display, and the meanings of commands to create that display; WEG works by combining code from its Java library, which we expect to be highly reusable.

Part of a DUCK specification is shown below. Its three sections are for webpage `classes`, interface `windows`, and `agents` to manipulate webpages and windows. For example, `[HPG]` is the homepage class, `(Bkg := pmarb)` means its background is `pmarb` (pink marble); `(Fr := HpFr)` means it uses the frame `HpFr`; and `(Tpl := HPTpl)` means it uses the template `HPTpl`. Under `windows`, the line beginning `[MAIN : HPG, TAT]` declares a window named `MAIN` that displays `HPG` and `TAT` webpage classes; the attribute `(pers)` means this window persists. The `->` arrow in front of `[HPG]` means that it is the root of the website, and the arrow in front of `[MAIN]` means it is the initial display in the user interface.

Items under the `agents` heading define editor and subeditor agents. For example, the line that begins `->PfHPg(MAIN)` declares the agent `PfHPg`, attached to the `MAIN` window, creating a directory `DirName` and homepage `Name`; the initial arrow indicates that this is the master agent. The notation `[[_]]` means the indicated item is actually created, while {_} means the required content is input from a popup window, {{_}} means it is input from a previously prepared file, and `:= ^` means the information is passed down from a parent node. `Goal.fol` indicates that `Goal` is expressed in first order logic, `Alg.obj` indicates that `Alg` is an OBJ file.

```
classes{
->[HPG](Bkg := pmarb)(Fr := HpFr) (Tpl := HPTpl)
  [TAT](Bkg := tat)  (Fr := TatFr)(Tpl := TatTpl)
  [EXP](Bkg := ymarb)(Fr := EFr)  (Tpl := ExTpl)
  [TUT](Bkg := ymarb)(Fr := EFr)  (Tpl := TutTpl)
  [OBJ](Bkg := brain)(Fr := OFr)  (Tpl := OTpl)
  [LG] (Bkg := brain)(Fr := OFr)  (Tpl := LgTpl) }
windows{
->[MAIN : HPG, TAT](pers)
  [EXP  : EXP, TUT]
  [OBJ  : OBJ, LG]
  [AUX  : HPG, TAT] }
agents{
->PfHPg(MAIN) :: [[{DirName}]] <{Name}>
    <Tpl>({Title} {SubTitle} {{Goal}}.fol
    {{Spec}}.obj {{Body}} [[TatPg]]((Name := ^)
    (Goal := ^)(Spec := ^)) )
  TatPg(TAT) :: <Name := ^><Tpl>([{Rule}]^*
    {{ExPg}}.htm (Obj := [[Alg]].obj))
  ExPg(EXP)  :: <Name := ^>
    <Tpl>((Name := ^)(Goal := ^)(Spec := ^)) }
```

We omit the agent action specifications, but note that they follow the style of hidden algebra [9]. The specification of logical systems as sets of conditional equations is well known and also omitted; e.g., see [7].

4. Semiotics, Narratology and Design

Since we aim to help ordinary software engineers, it is important to make specifications and proofs as understandable as possible. This requires better user interfaces than are currently available. Since interfaces consist of signs, often complex signs constructed hierarchically out of other signs, it is reasonable to think that a rigorous theory of signs would provide a scientific basis for interface design. This is a major motivation for **algebraic semiotics**, which uses algebraic specification to study complex signs and the quality of their representations [6]. Semiotic representations are useful not only in interface design, but also in automating tool development, e.g., in WEG.

Constructing non-trivial proofs usually requires exploring many misconceptions and errors, some of which may be very subtle. Published proofs bury all this, exhibiting only the tricks that finally overcame the obstacles, without showing why those tricks were needed. While human proofs often leave out details to make the main ideas clearer, mechanical proofs generally bury proof structure in a mass of trivial detail, since they lack features to exhibit large grain proof structure and motivation. We suggest making proofs more understandable and even interesting by restoring the local conflicts that motivate difficult proof steps, including failed proof attempts.

Another resource is the theory of stories. Labov [11] says oral narratives have a precise structure, involving a sequence of "narrative clauses" describing events whose order corresponds to their order in the story, interleaved with "evaluative material" which evaluate the events; there are also optional opening and closing sections. Labov's work was the major motivation for our linking explanation pages to tatami pages; further study of how evaluative material connects to narrative material might give further ideas for structuring proof motivation. The study of embedded stories, flashbacks, and so on could also be useful in making proofs easier to understand.

Aristotle said "*Drama is conflict*." Perhaps proofs can be made more understandable and interesting, by restoring the conflicts that motivate difficult proof steps, in subProofWebs recording failured subproofs, and in links to interactive applets and other verbal and graphical explanations. Syd Field's screenwriting books [2] suggest a precise but naive structure: Hollywood plots should have three acts, for setup, conflict, and resolution, of 30, 60, 30 minutes, with "plot points" to move action from one act to the next. Proofs that are structured this way might be easier to understand.

4.1. Sign Systems and Semiotic Morphisms

ProofWebs, tatami pages, and status windows are signs. An insight of Saussure [14] that every sign should be considered part of a system of interrelated signs. The material in this section, taken from [6], gives a rigorous foundation for discussing the structure of such representations.

Definition 1: A **sign system** S consists of:

1. a set S of **sorts** for signs, not necessarily disjoint;
2. a partial ordering on S, called the **subsort** relation and denoted $\leq$;
3. a set V of **data sorts**, for information about signs, such as colors, locations, and truth values;
4. a partial ordering of sorts by **level**, such that data sorts are lower than sign sorts;
5. a set C_n of level n **constructors** to build level n signs from signs at levels n or less, and written $r\colon\ s_1...s_k d_1...d_l \to s$ to indicate that its ith argument must have sort s_i, its jth parameter data sort d_j, and its result sort is s (constants $c\colon\ \to s$ are also allowed);
6. a **priority** (partial) **ordering** on each C_n;
7. some relations and functions on signs; and
8. a set A of **axioms**, constraining possible signs.

$\square$

A sign system is really an abstract data type [8, 7] enriched with levels and priority; we exploit this by specifying sign systems in OBJ3 in Appendix A.

We now consider the movement (mapping, translation, interpretation, representation) of signs in one system to signs in another. For example, tatami pages and status windows result from translating ProofWebs; and possibly proof structures that map into the "Syd Field" sign system would be easier to understand.

Definition 2: Given sign systems S_1, S_2, a **semiotic morphism** $M\colon\ S_1 \to S_2$, from S_1 to S_2, consists of the following partial functions (all denoted M):

1. sorts of $S_1 \to$ sorts of S_2,
2. constructors of $S_1 \to$ constructors of S_2, and
3. predicates and functions of $S_1 \to$ those of S_2,

such that

1. if $s \leq s'$ then $M(s) \leq M(s')$,
2. if $c\colon\ s_1...s_k \to s$ is a constructor (or function) of S_1, then (if it is defined) $M(c)\colon\ M(s_1)...M(s_k) \to M(s)$ is a constructor (or function) of S_2, and
3. if $p\colon\ s_1...s_k$ is a predicate of S_1, then (if defined) $M(p)\colon\ M(s_1)...M(s_k)$ is a predicate of S_2.

More generally, semiotic morphisms can map source system constructors and predicates to target system terms. $\square$

A semiotic morphism $S_1 \to S_2$ gives representations in S_2 for signs in S_1. If we know how a semiotic morphism maps constructors, then we can compute how it maps complex signs. For example, if $M(a) = a', M(b) = b', M(c)(x,y) = c'(x, y+1)+1$, and $M(f)(x,y) = x+y+1$, then $M(c(a, f(3,b))) = c'(a', b'+5)+1$. The set of semiotic morphisms between two fixed sign systems is ordered in various ways by how much structure is preserved; see [6] for details. The specification of interfaces as semiotic morphisms is used by WEG in generating editor interfaces.

Fauconnier and Turner [1] study the *blending* of (what they call) conceptual spaces, and suggest blending is a fundamental cognitive operation, with applications to metaphors, grammar, and much more. Many examples of blending can be found in interface design as well; e.g., the status window is a blend of structures for truth status and subproof ownership. The meaning of the word "proofweb" is a blend of the meanings of its constituent words. [6] gives a rigorous approach to blending through the *category* of sign systems with semiotic morphisms; this is an *ordered category* because of the ordering on morphisms. Category theory suggests that a blend should be a *pushout*; however, special kinds of ordered category and pushout are needed.

4.2. Some Design Decisions

This section shows how design decisions can be justified by algebraic semiotics. The basic insight is that any TATAMI implementation can be seen as a morphism from the `TATAMI` sign system (as specified in Appendix A) to a web sign system.

Background images The major sorts of web page are for proof step, explanation, and proof score; each has a distinctive background. These are the three major sorts of argument to the constructor `PfOf_By` in the specification of Appendix A; these sorts have important lower level sign systems, and should be preserved by any TATAMI implementation.

Persistent popups Auxiliary information, including logic tutorials, lemma proofs, and proof score execution results appear in persistent popups, one for each major sort of information. This structure can be justified as above, but a more detailed specification than that in Appendix A is needed. As a bonus, the Netscape BACK button retrieves information previously seen in each window.

Separating form and content A uniform multi-frame layout helps avoid confusion among pages. The top "title" frame names the current ProofWeb and node; the "button" frame supports navigation; and a "display" frame shows content.

Uniform look-and-feel and consistent transition Popup windows provide auxiliary information; e.g., there is a fixed window for browsing proofs of lemmas. Each page has a button for return to the page where it was referenced. Only navigation buttons change the master window.

Mathematics Formulae are difficult in current HTML; although proposals exist, there is no standard. We use `gif` files, and a distinctive color (blue), since this is a quite distinct world.

As the TATAMI project progressed, several different versions of the Status Window were developed; some are shown in Figure 5. In retrospect, we can see that these do *not* admit semiotic morphisms from the ProofWeb sign system, because they fail to preserve certain key constructors;

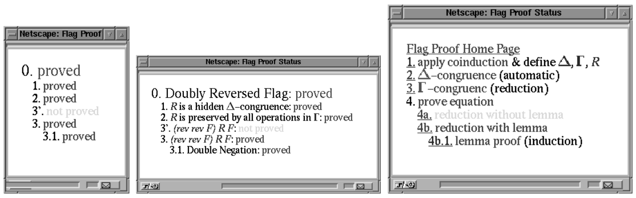

Figure 5: Screendumps of Three Status Window Versions

details are given in [10]. We view this as a strong confirmation of our approach.

Note that semiotic morphisms correspond to OBJ **views**, with partial morphisms given by views from sub-sign systems, as illustrated by the view STAT in Appendix A. It seems almost amazing that OBJ provides not only a simple, elegant way to specify sign systems, but also directly supports both blends and semiotic morphisms, through its facilities for parameterized programming [4].

5. Current Status and Further Research

We have used algebraic ADTs in several ways: to specify and verify object oriented systems (though this is not illustrated here – see [9]), to define logics, and to define sign systems and their morphisms, particularly in specifying website structure and guiding interface design. The formal notions of 2-doag and fan have helped in specifying and verifying the tatami protocol. Many aspects of this research are novel, including the tatami conventions, with its idea of proof as narrative, the tatami protocol, and the editor generator; semiotic morphisms appear to be especially novel [6, 10]. These developments also help support the automation of tools for distributed cooperative formal methods.

Several ProofWebs have been put on our website by hand, with applets, tutorials, and executable proofs, using BARISTA to access an OBJ server at UCSD. We have proved correctness of the tatami protocol using hidden algebra, and will soon add this to our website. A prototype of KUMO is available from a linked website, along with several ProofWebs generated by it; these are essentially identical to those generated by hand. WEG is still at an early phase of development.

We have had significant experience with people reading our proofs over the web, and the feedback has been positive. We have also done some crude experiments using a partial implementation of the tatami protocol; these suggest that future work should focus on database issues, especially for specifications. Our experience with the current KUMO prototype has suggested many improvements, and we are now undertaking a thorough redesign and reimplementation; in particular, the new version will support informal and semi-formal proofs.

An interesting new idea is to extend algebraic semiotics by using hidden algebra instead of just initial algebra semantics; then *dynamic* aspects of interface design can be captured. We also wish to explore using popup avatars for warnings, explanations, etc., perhaps following some ideas from [16].

Acknowledgments We thank Razvan Diaconescu for help with ordered category theory, Eric Livingston for discussions on social aspects of proof, and Kokichi Futatsugi for generous support and encouragement.

References

[1] G. Fauconnier and M. Turner. Conceptual projection and middle spaces. Technical Report 9401, University of California at San Diego, 1994. Dept. of Cognitive Science.

[2] S. Field. *Screenplay: The Foundations of Screenwriting*. Dell, 1982. Third edition.

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

[4] J. Goguen. Principles of parameterized programming. In T. Biggerstaff and A. Perlis, editors, *Software Reusability, Volume I: Concepts and Models*, pages 159–225. Addison Wesley, 1989.

[5] J. Goguen. Formality and informality in requirements engineering. In *Proceedings, International Conference on Requirements Engineering*, pages 102–108. IEEE Computer Society, April 1996.

[6] J. Goguen. Semiotic morphisms. Technical Report CS97–553, UCSD, Dept. Computer Science & Eng., 1997. Early version in *Proc., Conf. Intelligent Systems: A Semiotic Perspective, Vol. II*, ed. J. Albus, A. Meystel and R. Quintero, Nat. Inst. Science & Technology (Gaithersberg MD, 20–23 October 1996), pages 26–31.

[7] J. Goguen. *Theorem Proving and Algebra*. MIT, to appear.

[8] J. Goguen and G. Malcolm. *Algebraic Semantics of Imperative Programs*. MIT, 1996.

[9] J. Goguen and G. Malcolm. A hidden agenda. Technical Report CS97–538, UCSD, Dept. Computer Science & Eng., May 1997. Early version in *Proc., Conf. Intelligent Systems: A Semiotic Perspective, Vol. I*, ed. J. Albus, A. Meystel and R. Quintero, Nat. Inst. Science & Technology (Gaithersberg MD, 20–23 October 1996), pages 159–167.

[10] J. Goguen, A. Mori, and K. Lin. Algebraic semiotics, ProofWebs and distributed cooperative proving. In Y. Bartot, editor, *Proceedings, User Interfaces for Theorem Provers*, pages 25–34. INRIA, 1997. Sophia Antipolis, 1–2 September 1997.

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

[12] E. Livingston. *The Ethnomethodology of Mathematics*. Routledge & Kegan Paul, 1987.

[13] Luqi and J. Goguen. Formal methods: Problems and promises. *IEEE Software*, 14(1):73–85, 1997.

[14] F. Saussure. *Course in General Linguistics*. Duckworth, 1976. Translated by Roy Harris.

[15] D. Shand. *Proof Diagrams and Term Rewriting with Applications to Computational Algebra*. PhD thesis, University of St Andrews, School of Mathematical and Computational Science, 1997.

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

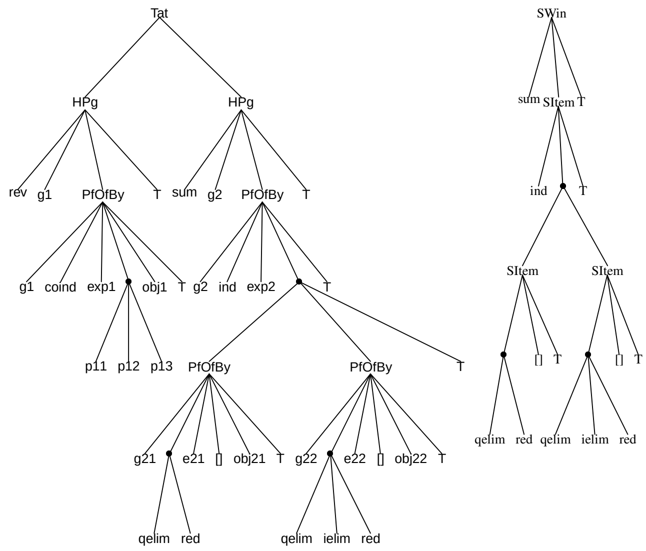

Figure 6: Structures of Two Typical ProofWebs and a Status Window

A. ProofWeb and Status Window ADTs

Below is a partial specification for ProofWebs in OBJ, with the structure of two proofs that are on the web; their trees are shown in Figure 6.

```
obj TATAMI-BASE is sorts Pf Stat .
  dfn Name is QID .     dfn Rule is QID .
  dfn LPf is LIST[Pf].  dfn LRule is LIST[Rule].
  dfn Obj is QID .      dfn ExPg is QID .
  ops proved unknown untrue : -> Stat .
endo
obj TATAMI-PF is pr TATAMI-BASE .
  sort HPg .
  dfn Goal is QID .  dfn Rule is QID .
  dfn LPf is LIST[Pf].
  op HPg_[___] : Name Goal Pf Stat -> HPg .
  op PfOf_By_[___]_ :
    Goal LRule ExPg LPf Obj Stat -> Pf .
endo
obj TATAMI is pr TATAMI-PF .
  sort Tatami .
  dfn LHPg is LIST[HPg].
  op Tat  : LHPg -> Tatami .
endo
obj TESTAMI is pr TATAMI .
  op exp : Nat -> ExPg .   op pf  : Nat -> Pf .
  op g   : Nat -> Goal .   op obj : Nat -> Obj .
  let p21 : Pf = PfOf g(21) By('qelim 'red)
              [exp(21) (nil) obj(21)] proved .
  let p22 : Pf = PfOf g(22) By('qelim 'ielim 'red)
              [exp(22) (nil) obj(22)] proved .
  let p1 : Pf = PfOf g(1) By('coind)
             [exp(1) (pf(11) pf(12) pf(13))
              obj(2)] proved .
  let p2 : Pf = PfOf g(2) By('ind)
             [exp(2) (p21 p22) obj(2)] proved .
  let rev : HPg = HPg('rev)[g(1) p1 proved].
  let sum : HPg = HPg('sum)[g(2) p2 proved].
  let t   : Tatami  = Tat(rev sum).
endo
red sum .  red t .
```

Viewing `TATAMI` as a sign system, its top level signs have sort `tatami` with the primary constructor `Tat`, which takes one argument of sort `LHPg`, which in turn has as its primary constructor the concatenation operation for lists of `HPg`s, proof homepages. `HPg`s have primary constructor `HPg`, and technically are third level signs, but since there is just one unary constructor, let's just agree to call them second level. There is also a secondary constructor for `LHPg`, namely the empty list, denoted `[]`. The arguments of `HPg` are the third level signs, with sorts `Name`, `Goal`, `Pf` and `Status`. The most interesting of these is `Pf`, for proofs, with primary constructor `PfOfBy`, which has `LPf`, for lists of proofs, as an argument. Therefore `Pf` is a recursive sign sort; it is the sort for what we have called tatami pages. The fourth level sign sorts are `LRule`, `ExPg` and `Obj`, for rule list, explanation page, and OBJ code, respectively.

All the above OBJ code runs, as does that below, with results exactly (terms for) the trees shown in Figure 6; the dotted nodes in this figure indicate the concatenation operation for lists. We now specify TATAMI status windows, with the sum proof as a test case.

```
obj STATWIN is pr TATAMI-BASE .
  sorts SWin SItem .
  dfn LSItem is LIST[SItem] .
  op Swin_[_]_  : Name   SItem Stat -> SWin .
  op SItem_[_]_ : LRule LSItem Stat -> SItem .
endo
obj TESTSWIN is pr STATWIN .
  let i11 = SItem('qelim 'red) [nil] proved .
  let i12 = SItem('qelim 'ielim 'red) [nil]
            proved .
  let i1  = SItem('ind) [i11 i12] proved .
  let w1  = Swin('sum)  [i1] proved .
endo
red w1 .
view STATV from TATAMI-PF to STATWIN is
  sort HPg to SWin .  sort Pf  to SItem .
  sort LPf to LSItem .
  var G : Goal . var LR : LRule . var E : ExPg .
  var O : Obj . var S : Stat .  var LP : LPf .
  var N : Name . var P : Pf .
  op (PfOf G By LR [E LP O] S)
    to (SItem(LR) [LP] S) .
  op (HPg N [G P S]) to (Swin N [P] S) .
endv
```

This view is defined for single ProofWebs (i.e., the `TATAMI-PF` spec) not systems of ProofWebs (the `TATAMI` spec), i.e., it is a *partial* semiotic morphism from `TATAMI` to `STATWIN`; also, `w1` is the image of `sum` under it.