# An Overview of the Tatami Project*

Joseph Goguen[a] and Kai Lin[a] and Grigore Roşu[a] and Akira Mori[b], and Bogdan Warinschi[a]

[a] Dept. Computer Science & Engineering, University of California, San Diego
9500 Gilman Drive, La Jolla CA 92093-0114 USA
phone: +1-858-822-1588, fax: +1-858-534-7029
email: {goguen,klin,grosu,bogdan}@cs.ucsd.edu

[b] Japan Advanced Instititute of Science and Technology, Hokuriku
1-1 Asahidai Tatsunokuchi Nomi Ishikawa, 923-1292 JAPAN
email: amori@jaist.ac.jp

This paper describes the Tatami project at UCSD, which is developing a system to support distributed cooperative software development over the web, and in particular, the validation of concurrent distributed software. The main components of our current prototype are a proof assistant, a generator for documentation websites, a database, an equational proof engine, and a communication protocol to support distributed cooperative work. We believe behavioral specification and verification are important for software development, and for this purpose we use first order hidden logic with equational atoms. The paper also briefly describes some novel user interface design methods that have been developed and applied in the project.

## 1. Introduction

The Tatami system design was motivated by three main goals: (1) verify distributed concurrent software; (2) provide a distributed environment for cooperative work; and (3) produce proofs that are easier to read. This system differs from related systems with which we are familiar, in many or all of the following respects:

1. it is rigorously based on an advanced version of hidden algebra allowing first order sentences with equational atoms interpreted behaviorally;
2. design is separated from validation, with a distinct language for each activity;
3. distributed cooperative work is supported;
4. there is a distributed multi-project database;
5. a specialized protocol maintains database consistency in the presence of semi-reliable internet communications;

*The research reported in this paper has been supported in part by the CafeOBJ project of the Information Promotion Agency (IPA), Japan, as part of its Advanced Software Technology Program, and by National Science Foundation grant CCR-9901002.

6. a range of formality is supported from full mechanical proofs to informal "back of envelope" arguments, using fuzzy logic for the confidence values of assertions;
7. web-based interactive documentation is automatically generated for proofs;
8. there are links to online tutorials on background material;
9. recent web and net technologies are heavily used, including secure HTTP, XML [40], XSL [41], SSL [39], and cgi; and
10. many user interface design decisions have been rigorously based on principles from cognitive psychology, narratology, semiotics, etc.

Large software projects have multiple workers, often at multiple sites with different schedules. Therefore it is difficult to share information, coordinate tasks, and maintain consistent code, documentation and requirements. We seek to solve some of these problems by building tools to support distributed cooperative work over the web. Our current emphasis is on proving properties of distributed concurrent systems, such as communication protocols. Although full formal verification is an option, the most practical approach is probably to 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 tools and for using them.

We focus on behavioral specification and verification, because real software systems often do not satisfy their specifications strictly, but instead only satisfy them *behaviorally*, i.e., appear to satisfy them in all relevant situations. Our logic for this is hidden algebra [19,35]. Code is regarded as secondary, because it can be (relatively!) quickly written or even automatically generated from specifications that are sufficiently modular and detailed, and empirical studies show that little of software cost comes from errors in coding [3]. Therefore we focus on specification and verification at the design level, and avoid the ugly complications of programming language semantics.

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 explain and document proofs. Examining any mathematics book or paper, even one in logic, shows that mathematicians almost never write formal proofs in the strict sense of mathematical logic. The only proofs written this way are extremely simple illustrations of formal proof methods, not proofs of any genuine mathematical interest. This is because all but the simplest fully formal proofs are practically impossible to comprehend. Unfortunately, these are exactly the kind of proofs produced by mechanical theorem provers. In order to improve this dismal situation, we suggest that the motivation and structure of proofs should be made explicit, and that relevant background and tutorial material should be integrated with proofs. These recommendations follow ideas from cognitive psychology, narratology and semiotics [11,12], as discussed further in Section 3. In particular, the structure of the Tatami system proof websites was designed using algebraic semiotics, which combines algebraic specification with social semiotics in the sense of [10]. The present paper is an extension, revision and amalgamation of work reported in [15,16] and other papers. Although this particular paper focuses on system architecture and user interface issues, for the convenience of interested readers, the bibliography attempts to list most of the papers produced by the project, including those concerning other aspects, such as the underlying formal logic. The latest information on the Tatami project can always be accessed via its URL, `www.cs.ucsd.edu/groups/tatami/`.

## 2. Tatami System Design

This section describes the Tatami System, including its central component, a proof assistant and website generator component KUMO[2], its specification language and underlying hidden logic (Section 2.1.1), its command language (Section 2.1.2), database (Section 2.2), and its specialized communication protocol (Section 2.3). Some further implementation details are given in Section 2.4.

### 2.1. KUMO

KUMO executes proof commands, and generates websites that document its proofs. Users of Tatami mainly interact with KUMO through its database management interface and its two specialized languages, BOBJ for (behavioral) specifications, and DUCK, which has commands for both proof execution and proof display. KUMO has two user modes, called "browser" and "owner." Browser mode provides access to all of the public information, while owner mode is used for creating and editing specifications and proofs. A typical session goes as follows (see Figure 1):

1. Choose a project.
2. Optionally, introduce a new specification.
3. Select an existing unsolved proof task, or introduce a new proof task.
4. Enter or edit commands in a DUCK text, and then execute it; this produces or updates the corresponding proof website.
5. View the results on a web browser.
6. Repeat from step 2.
7. When done, save your work; all subtasks of the selected task are then entered into the Tatami database.

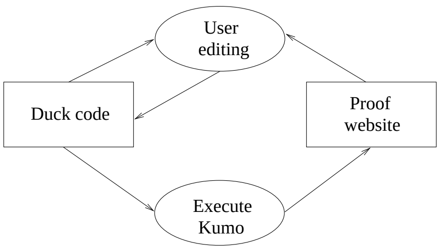


Figure 1. The Edit-Execute-Browse Cycle

While this edit-execute-browse cycle (again see Figure 1) might seem old-fashioned to some readers, empirical studies [29] and our own experience have found that the current fad for direct manipulation interfaces for theorem proving systems is actually counter-productive for complex proofs, although it may have some value when used on small

[2]This is a Japanese word for spider.

proofs as a pedagogical aid, e.g., for beginning students of formal logic. The problem is that the data structures that underlie proofs of any real interest are too large. Note also that the specification may also undergo evolution as a part of the cycle described above.

#### 2.1.1. Hidden Algebra and BOBJ

The Tatami project uses an approach to behavioral specification and verification called **hidden algebra**, which extends standard many sorted algebra by distinguishing between "visible" sorts used to model data, and "hidden" sorts used to model states, as originally proposed in [9] and elaborated in many subsequent publications. This framework provides simple and natural ways to define behavioral equivalence of states, behavioral satisfaction of properties, and behavioral refinement of specifications. Standard equational deduction generalizes with small changes, but more powerful inference rules are needed for most interesting proofs. In particular, we have been developing a series of increasingly powerful **coinduction** rules [33–35], which greatly extend deductive power, and which in practice yield conceptually simple and highly mechanizable proofs; the most recent of these is called **circular coinductive rewriting** – see [34]. Recent research has shown that it is impossible to give a complete recursively enumerable set of inference rules for hidden algebra [4], so there cannot be any final resting point on this quest. For details on hidden algebra, see [18–21,34,35].

Hidden algebra handles 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 (as in logic programming). In comparison with process algebra and transition systems, hidden algebra allows better control over the data involved, and admits equations with operations having multiple visible and hidden parameters, thus greatly extending expressive power.

Behavioral logic is a diverse research area containing many approaches, including the original hidden algebra of [9] and subsequent improvements in [14,19,18], the coherent hidden algebra of Diaconescu [6,7], the observational logic of Bidoit and Hennicker [1,2,24], and a new generalization of hidden algebra that tries to treat all these variants in a uniform way [35,21]. 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 [4]. Padawitz's "swinging types" are a powerful but less closely related approach [31].

The BOBJ (for Behavioral OBJ) hidden algebraic specification language extends the classical algebraic specification language OBJ [23] to behavioral properties, and can be considered a dialect of the CafeOBJ specification language [7]. Like CafeOBJ, it supports both classical and hidden algebraic specification; in addition, it supports behavioral operations with multiple hidden arguments, cobases, and circular coinductive rewriting. A very brief introduction to BOBJ appears in [34]; the system has recently been implemented in Java. The Tatami system and Kumo extend the logic of BOBJ by allowing first order sentences with behavioral equations as atoms, and induction for the initially defined data types.

#### 2.1.2. Duck

Duck contains the proof command language for Kumo, combining proof rules for hidden algebra and first order logic, including the following:

- elimination rules for $\forall$, $\exists$, $\wedge$, $\vee$, $\Rightarrow$, and $\neg$;
- Skolemization and lemma introduction;
- case analysis, modus ponens, proof by contradiction and substitution;
- term rewriting and equational deduction;
- induction; and
- coinduction.

DUCK also has a sublanguage for proof display, which provides the URLs and sets the parameters used when KUMO produces the XML file containing the tree structure of the proof. For example, when conjunction elimination is applied to the goal $T \vdash_{\Sigma} A \wedge B$, the two new subgoals, $T \vdash_{\Sigma} A$ and $T \vdash_{\Sigma} B$, are added to the XML file. KUMO then uses an XSL style file to generate HTML text for the proof; at this stage, links to tutorials, machine proof scores, and explanation pages are added, as described in more detail in Appendix B. DUCK also has the capability to place several proofs on a single page, and it automatically provides appropriate cross-references to other proofs used, such as lemmas.

### 2.2. The Tatami Database

Any practical implementation of formal methods must support bookkeeping for proof tasks, subtasks, dependencies, and specifications. Our system does this using a distributed database, the coherence of which is maintained by a specialized protocol. Each worker has a local database which is conceptually divided into the three components that are described in the three subsections below.

#### 2.2.1. The Project Database

The Project Database maintains a list of projects, with their members and leaders. A project is started by placing it in the database; the worker who does so becomes the leader of that project. Information at the project database level can only be modified in owner mode by the leader of a project.

#### 2.2.2. The Specification Database

The Specification Database has two major parts, one for specifications of data which are used for values, the other for abstract (software) machine specifications. A number of binary relations may be defined on specifications, including the following:

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

There are also some important relations that hold among these relations, for example that enrichment implies refinement, and that equivalence implies enrichment.

#### 2.2.3. 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. All this is handled by a data

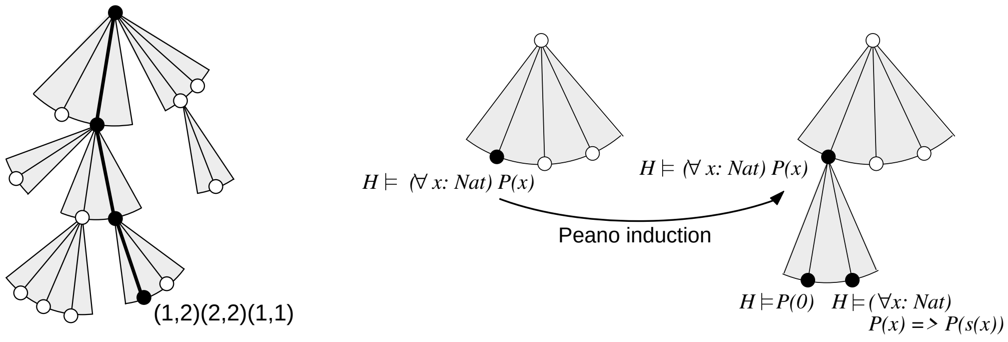


Figure 2. 2-occurrences and Peano induction

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 2, 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 the source node.

Navigation through 2-doags is accomplished with **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, as illustrated in the left side of Figure 2. A formal description of 2-doags is given in Appendix C. These structures may be further specialized 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 [8] to evaluate the expressions. The truth value 1 means that there is a formal proof, while 0 means that there is a formal disproof. (The logic of [8] differs from the usual fuzzy logic in using multiplication for conjunction, instead of maximum.)

- 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 side of Figure 2 shows a proofdoag update that applies Peano induction for the natural numbers.

### 2.3. The Tatami Protocol

A problem with distributed databases 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 in a local database unless $p_2$ has already been entered. Inconsistencies can similarly arise when items are deleted. The

Tatami protocol maintains the consistency of the Tatami database, taking account of the following situations:

1. If the owner of an item decides to delete it, then ownership is passed to a randomly chosen worker who is using it. A message is broadcast to all involved databases to make this transfer public, and the item is deleted from its original location only after acknowledgments are received from all of them.
2. When something is missing from a just received message, the message must be retained, but not installed in the database until the missing part arrives.

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 is one of 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).
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 the Tatami protocol with respect to a communication medium that can lose or duplicate data [22], and we have implemented it using the IP internet protocol.

### 2.4. Some Implementation Details

Figure 3 is an overview of the Tatami system architecture. Its most important components are the KUMO website generator and proof assistant, the Tatami database, the Tatami protocol, the BARISTA[3] proof engine server, one or more proof engine (especially some version of OBJ), and a standard web browser. The Tatami database is implemented using the mSQL relational database system, and BOBJ and DUCK are implemented using Java technologies, including JavaCC for parsing. When viewing generated webpages in browser mode, a modified HTTP server is used, while owner mode uses a secure HTTP server, built using SSL.

The XML files generated by KUMO are passed to a processor that uses an XSL style file to generate the HTML pages that are actually seen by users. Figure 4 is a UML use case diagram showing in more detail the actions that take place when a DUCK text is processed.

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

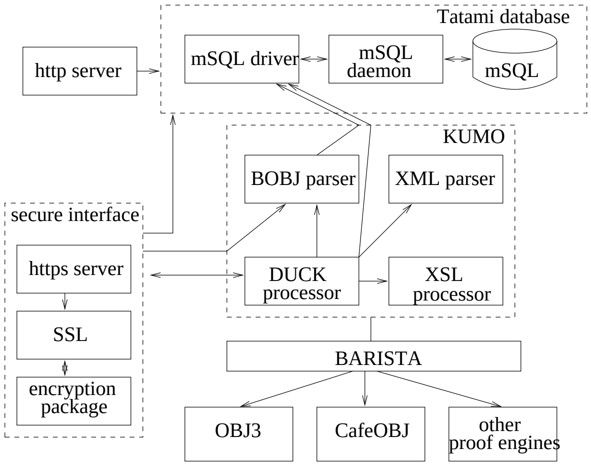


Figure 3. Tatami Implementation Overview

The Tatami project does not seek 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 using the BARISTA server. Our current prototype wraps an OBJ3 proof engine, but BARISTA can easily be adapted to other proof engines, especially CafeOBJ and (very soon, we hope) BOBJ.

This modular architecture was designed to facilitate use of a variety of logics. The following components are ogic-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 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.

## 3. User Interface Design

Because we aim to help ordinary software engineers, user interface issues are of great importance for the Tatami project. Cognitive science, semiotics (the theory of signs), narratology (the theory of stories) and even cinema, have influenced our designed decisions, as briefly explained below, and in much more detail in [11]. Of particular interest is algebraic semiotics, a novel technique that provides systematic ways to evaluate the quality of user interfaces, such as proof presentations.

### 3.1. Algebraic Semiotics

An important insight attributed to Ferdinand de Saussure [37] 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 formalize the intuitive notion of a **sign system** using the tools of algebraic specification [17], as a loose algebraic theory (consisting of a signature and some axioms) plus further structure, such as

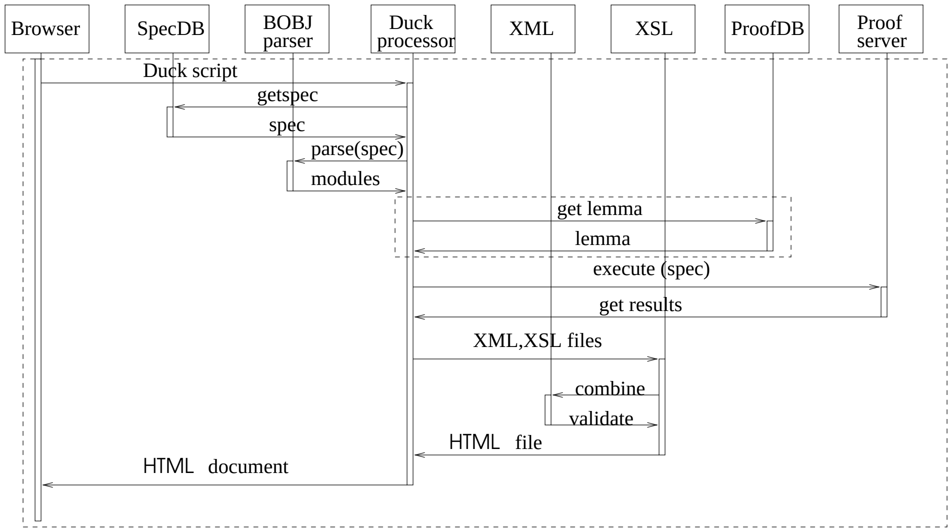


Figure 4. Use Case Diagram for DUCK Execution

constructors, a hierarchy of levels for signs, and priorities among constructors. A recent insight that is important for the Tatami project is that *representations* can be seen as translations or maps from one sign system to another [11]; it is similarly natural to formalize these translations using the notion of *theory morphism* from algebraic specification [13]. But examples in our application domains show that maps between sign systems do not in general fully preserve structure, and in particular, must involve *partial* functions. These considerations motivate the definition of **semiotic morphism** that is given in [11]. The key point for present purposes is that the *quality* of a representation can be examined in terms of what structure is preserved by its semiotic morphism.

User interfaces are of course prime examples of representations, and it is therefore natural to study them using semiotic morphisms which map from the sign system of the given information to that of its display; the quality of the interface is then measured by the quality of its semiotic morphism. Details, some of which are surprisingly technical, may be found in [12] and [11].

### 3.2. Narratology

Typical proofs in modern mathematics hide the often considerable conflicts that were involved in their construction, since finding a non-trivial proof usually requires exploring many misconceptions and errors, some of which may be very subtle. We claim that proofs can be made more understandable, and even more interesting, if the conflicts that motivate their difficult steps are integrated into their structure, instead of being ignored. As Aristotle said, "*Drama is conflict.*"

An important resource for our work has been the theory of oral narratives developed by William Labov [25], who showed that these have a precise structure, involving a sequence of so-called "narrative clauses" which describe events (the default ordering of which corresponds to their order in the story), interleaved with "evaluative material" which *evaluates* the events, in the sense of relating them to socially shared values. There is also an optional opening "orientation" section, giving necessary background for understanding the story, such as time and place, as well as an optional closing section containing a "moral" or summary for the story; see also [26,27]. The influence of these ideas on our proof website design conventions is discussed in in the next section.

### 3.3. Proofwebs and the Tatami Conventions

Proof representations can be described at several levels of abstraction. What we call a **concrete proofweb** describes our abstract structure for formal proof steps combined with informal elments; it has a tree structure with proof steps as nodes, decorated with pointers to relevant background and explanation material. An **abstract proofweb** has a similar tree structure, but omits the background and explanation material, as well as details of proof rules; only the information needed to support cooperative work is retained. Finally, a **display proofweb** is a graphical representation, consisting of windows, colors, navigation buttons, etc. Its content is the information provided by the corresponding concrete proofweb.

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 [32,38]. We have developed the following **tatami conventions** as style guidelines for the proof websites generated by Kumo, which we are calling display proofwebs:

1. **Tatami pages** are the most important constituents of proof websites, since they contain the inference rule applications that are the heart of proving. Tatami pages appear in a fixed master window, an example of which is shown on the right side of Figure 5; each page should have a relatively small number of proof steps (about seven non-automatic rules per page works well).
2. Tatami pages can be browsed in a "narrative" 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).
3. A prover-supplied informal explanation page is linked to each tatami page, discussing the proof concepts, strategies, obstacles, etc. for that page; these can have graphics, applets, and of course text; they appear in a dedicated persistent popup window.
4. Major proof parts, including lemmas, each have their own homepage, which can include graphics, applets, and text; these appear in the same window as their tatami pages; see the left side of Figure 5. There is a dedicated persistent popup window for lemmas which has the same structure as the master window.
5. Major proof parts can have their own "closing" webpage to sum up results and lessons; these appear in the same window as their tatami pages.
6. 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 also have their own dedicated persistent popup window.

7. A detailed proof score is generated for each proof page; proof readers can view these on another dedicated persistent popup, and can request their execution on a proof engine, with result displayed in the same window as the proof score.
8. Each kind of webpage has its own distinctive background and frame; the frame provides navigation buttons that are appropriate for that kind of page.
9. A menu of open subgoals is placed on each homepage, and error messages are placed on the most appropriate pages. A summary of this information also appears in the status window, which is a specialized popup that summarizes information about the current status of a proof or part thereof.

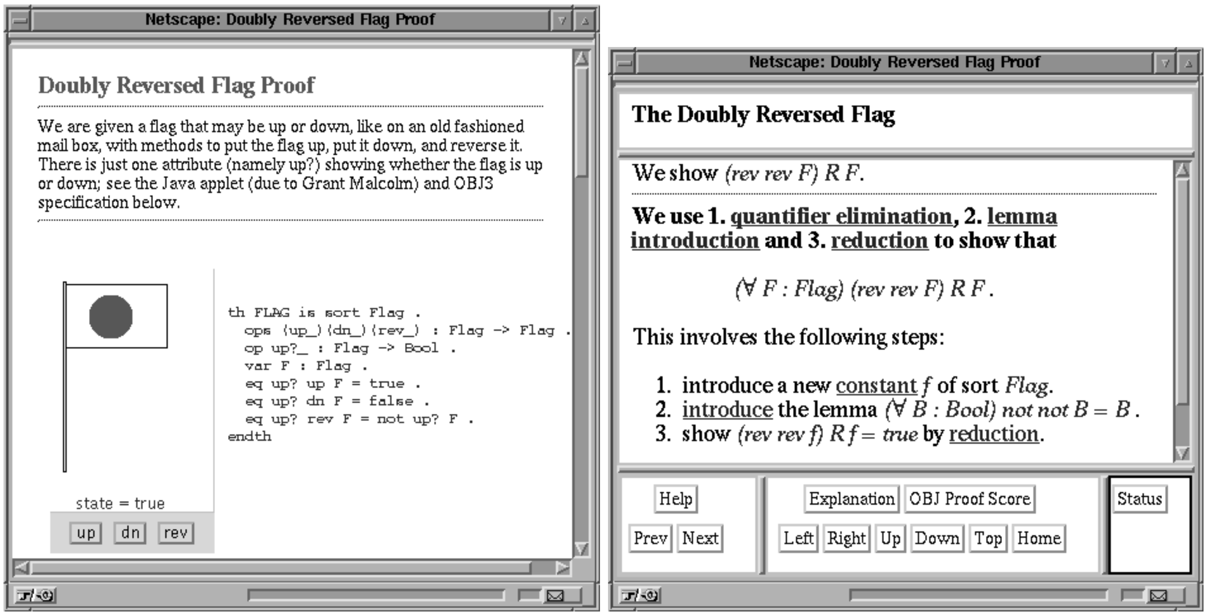


Figure 5. A Typical Tatami Homepage and Proof Page

These conventions have the effect of integrating proofs with the information that is needed to motivate, understand and debug them, and should therefore make proofs easier to do and easier to understand.

### 3.4. Justifying Design Decisions

This section applies the techniques discussed in Sections 3.1 and 3.2 to some design decisions for the proof websites generated by KUMO. We first justify the tatami conventions using the same enumeration as in Section 3.3; many of these justification draw on narratology.

1. Limiting the number of non-automatic proof steps on tatami pages to approximately 7 is consistent with known limitations of human cognitive capacity [30].
2. The idea of a giving a "narrative" order to tatami pages comes from the theory of stories [25]; the idea of including obstacles comes from Campbell [5] and others.

3. Attaching prover-supplied informal explanation pages to proof pages was suggested by the close connection between narrative clauses and evaluative material in stories [25]; the evaluative material provides the motivation needed for important steps in the proof, by relating them to values shared among the community of provers. Placing these in a separate window parallels the syntactic structure used in stories.
4. Using homepages for major proof parts is motivated by the opening "orientation" sections in Labov's theory of story structure [25]; homepages appear in the same window as their tatami pages, because they are part of the same narrative flow.
5. The optional closing webpages for proofs are also inspired by Labov's theory of story structure [25], and they too appear in the same window as proof pages, again because they are part of the same narrative.
6. Linking proof steps to tutorial pages can be motivated by some connectionist theories that concepts are organized as linked structures [36].
7. 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 [28]; however, proof readers can still view them, and even execute them on a proof server.
8. The justification for giving each kind of webpage a different background and a different frame is discussed below.
9. Open subgoals are very important to provers when they read a proof; they are the leaf nodes of the current proof tree, and hence form a list in a natural way.

The remaining design decisions are justified using algebraic semiotics. The basis for these arguments is that any display to users of information in the system can be seen as a semiotic morphism from the sign system for concrete proofwebs to that of display proofwebs.

- **Windows:** The main contents of an abstract proofweb are its proof steps, informal explanations, tutorials, and mechanical proof scores. These four sorts of item are also the main contents of abstract proofwebs, and their preservation is very basic to the quality of this representation. This basic classification into four sorts is reflected in our choice to use a separate window to display each of them. Because Tatami pages are the main constituent of concrete proofwebs, theirs is the master window. Explanation pages have a persistent popup window triggered by a button on the master window, and dismissable by the proof reader with a button in its frame; tutorial and proof score pages are treated similarly, but have different colors and textures. There is also a persistent dismissable popup for lemmas, which has the exactly same structure as the master window.

- **Backgrounds:** Each kind of window has its own distinctive background: tatami pages have a tatami mat background, explanation pages have a pink marble background, tutorial pages have a yellow marble background, and proof score pages have a blue raindrop background. This again reflects the importance and distinctness of four main components of concrete proofwebs, and is justified by the importance of their preservation.

- **Frames:** A similar argument holds for frames. A top "title" frame contains the name of the current display proofweb. A "button" frame on the bottom supports navigation; these are slightly different for each kind of page, but have a uniform look and feel. The third frame holds the content, proof pages for the proof steps. Each persistent window has its own fixed layout and frames, with a button that returns to the page where it was requested.

- **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.

Some additional, more precise, applications of semiotic morphisms to the user interface of the Tatami system are described in [12]. For example, we were able to show 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 now taught at UCSD uses ideas from algebraic semiotics (see `www.cs.ucsd.edu/users/goguen/courses/271/`), and we believe that much more can be done along these lines.

## 4. Conclusions and Future Research

The Tatami project has come a long way from its beginnings in 1996. Inspired by the ambitious goals of the CafeOBJ project of which it was a small part, it developed in a perhaps surprising diversity of directions, including theoretical foundations of behavioral verification for distributed concurrent systems (i.e., hidden algebra), web-based system development (the Tatami system itself), and user interface design (using algebraic semiotics). Although significant progress has been made, a very 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.

In addition, we have new ideas for generalizing algebraic semiotics to better handle dynamic interfaces by extending its foundation from classical algebraic semantics to hidden algebra. We also plan to use the Tatami system in teaching the UCSD graduate course on programming languages, which will no doubt 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 difficult applications like communication protocols.

**Acknowledgements**

We thank Prof. Kokichi Futatsugi for encouragement and support through the CafeOBJ project in Japan, Akiyoshi Sato for writing some daemons for an early prototype of the Tatami protocol, and Prof. Eric Livingston for discussions of social issues. We also thank the rapidly developing international community interested in behavioral specification and verification for encouragement.

**REFERENCES**


1. Gilles Bernot, Michael Bidoit, and Teodor Knapik. Observational specifications and the indistinguishability assumption. *Theoretical Computer Science*, 139(1-2):275–314, 1995. Submitted 1992.
2. Michael Bidoit and Rolf Hennicker. Observer complete definitions are behaviourally coherent. Technical Report LSV–99–4, Ecole Normale Superior de Cachan, 1999.
3. Barry Boehm. *Software Engineering Economics*. Prentice-Hall, 1981.
4. Sam Buss and Grigore Roşu. Incompletness of behavioral logics, 1999. To appear in *Proceedings, Coalgebraic Methods in Computer Science* (Berlin, Germany, 25–26 March 2000), Electronic Notes in Theoretical Computer Science, Volume 33, 2000.
5. Joseph Campbell. *The Hero with a Thousand Faces*. Princeton, 1973. Bollingen series.
6. Răzvan Diaconescu. Behavioural coherence in object-oriented algebraic specification. Technical Report IS–RR–98–0017F, Japan Advanced Institute for Science and Technology, June 1998. Submitted for publication.
7. 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.
8. Joseph Goguen. The logic of inexact concepts. *Synthese*, 19:325–373, 1968–69.
9. 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.
10. 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.
11. 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.
12. 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.
13. Joseph Goguen and Rod Burstall. Institutions: Abstract model theory for specification and programming. *Journal of the Association for Computing Machinery*, 39(1):95–146, January 1992.
14. 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.
15. Joseph Goguen, Kai Lin, Akira Mori, Grigore Roşu, and Akiyoshi Sato. Distributed cooperative formal methods tools. In Michael Lowry, editor, *Proceedings, Automated Software Engineering*, pages 55–62. IEEE, 1997.
16. Joseph Goguen, Kai Lin, Akira Mori, Grigore Roşu, and Akiyoshi Sato. Tools for distributed cooperative design and validation. In *Proceedings, CafeOBJ Symposium*.

Japan Advanced Institute for Science and Technology, 1998. Namazu, Japan, April 1998.

17. Joseph Goguen and Grant Malcolm. *Algebraic Semantics of Imperative Programs.* MIT Press, 1996.
18. Joseph Goguen and Grant Malcolm. Hidden coinduction: Behavioral correctness proofs for objects. *Mathematical Structures in Computer Science*, 9(3):287–319, June 1999.
19. Joseph Goguen and Grant Malcolm. A hidden agenda. *Theoretical Computer Science*, to appear. Also UCSD Dept. Computer Science & Eng. Technical Report CS97–538, May 1997.
20. 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 *Electronic Journal of Functional and Logic Programming*, MIT, 1999.
21. 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.
22. Joseph Goguen and Grigore Roşu. A protocol for distributed cooperative work. In Gheorghe Stefanescu, editor, *Proceedings, FCT'99, Workshop on Distributed Systems*, pages 1–22. Elsevier, 1999. (Iaşi, Romania). Also, Electronic Lecture Notes in Theoretical Computer Science, Elsevier Volume 28, to appear 1999.
23. 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.* Kluwer, 2000. Also Technical Report SRI-CSL-88-9, August 1988, SRI International.
24. 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.
25. William Labov. The transformation of experience in narrative syntax. In *Language in the Inner City*, pages 354–396. University of Pennsylvania, 1972.
26. Charlotte Linde. The organization of discourse. In Timothy Shopen and Joseph M. Williams, editors, *Style and Variables in English*, pages 84–114. Winthrop, 1981.
27. Charlotte Linde. *Life Stories: the Creation of Coherence.* Oxford, 1993.
28. Eric Livingston. *The Ethnomethodology of Mathematics.* Routledge & Kegan Paul, 1987.
29. 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.
30. 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.
31. Peter Padawitz. Swinging types = functions + relations + transition systems, 1999. `issan.informatik.uni-dortmund.de/~peter`. Submitted to *Theoretical Computer Science.*

32. Jenny Preece, Yovonne Rogers, et al. *Human-Computer Interaction.* Addison Wesley, 1994.
33. 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.
34. Grigore Roşu and Joseph Goguen. Circular coinduction, 1999. UCSD Technical Report; revised version submitted for publication.
35. 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.
36. David E. Rumelhart and J.L. McClelland, editors. *Parallel Distributed Processing.* MIT, 1986.
37. Ferdinand de Saussure. *Course in General Linguistics.* Duckworth, 1976. Translated by Roy Harris.
38. Ben Shneiderman. *Designing the User Interface.* Addison Wesley, 1997.
39. SSL 3.0 specification. `home.netscape.com/eng/ssl3/ssl-toc.html`.
40. XML™ (Extensible Markup Language) specification. `www.w3.org/XML/`.
41. XSL (Extensible Stylesheet Language) specification. `www.w3.org/TR/WD-xsl`.


## A. Sample Duck Code

A Duck text for proving correctness of the traditional array-with-pointer implementation of stack is given below. It divides into two parts, the first for proofs and the second for display. In addition to the main theorem, there are three lemmas. Each lemma is proved by induction, while 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 its built in tactics that require no user intervention, including reduction, the elimination rules for universal quantifiers, conjunction, and so on. 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, which has the URL `www.cs.ucsd.edu/groups/tatami/demos/array/`, and information about the latest version of Kumo can be obtained from the URL `www.cs.ucsd.edu/groups/tatami/kumo/`.

```
project: Stack
getspec: PTR+ARR.1.klin
relation:
   op _R_ : Stack Stack -> Bool .
   var I1 I2 : Nat .
   var A1 A2 : Arr .
   eq (s I1 || A1) R (s I2 || A2) =
          I2 == I1 and A1[I1] == A2[I1] and (I1 || A1) R (I1 || A2) .
   eq (0 || A1) R (0 || A2) = true .
[]
cobasis: {top, pop}
***************************************************
proof: <<Lemma1>>
```

```
    goal: (forall X  Y : Nat)
            ((s Y) <= X  implies (exists Z : Nat)(X = s Z  and  Y <= Z )) .
by: induction on X with scheme {0,s} []
*****************************************************
proof: <<Lemma2>>
    goal: (forall X  Y : Nat) ( s X <= Y  implies  X <= Y  ) .
by: induction on X with scheme {0,s} []
*****************************************************
proof: <<PutAndBar>>
    goal: (forall I J N : Nat  A : Arr)
             (  I <= J   implies  (I || put(N,J,A)) R (I || A) ) .
by: induction on I with scheme {0,s} []
*****************************************************
proof: <<StackThm>>
  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) ) .
by: coinduction []
****************************************************
display:
   title: "Lemma for Pointer Array Implementation of Stack"
<<Lemma1>>
   subtitle: "We prove a basic property of the order on natural numbers."
[]
***************************************************
display:
   title: "Lemma for Pointer Array Implementation of Stack"
<<Lemma2>>
   subtitle: "We prove a basic property of the order on natural numbers."
[]
***************************************************
display:
   title: "Key Lemma for Stack Implementation"
   gethomepage:  /home/klin/he/putandbarhome.html
<<PutAndBar>>
   subtitle: "We prove the key lemma for stack implementation."
[]
****************************************************
display:
   title: "Pointer Array Implementation of Stack "
   gethomepage: /home/klin/he/stackhome.html
   getspecexpl: /home/klin/he/stackspecexp.html
<<StackThm>>
   subtitle: "We show that array-with-pointer implementation of stack is correct."
   getexpl:  /home/klin/he/stackexp1.html
   <<StackThm.1>>
   subtitle: "We prove <math>R</math> is <delta/>-congruence."
   getexpl:  /home/klin/he/stackexp2.html
   <<StackThm.2>>
   subtitle: "We prove the main result."
   getexpl:  /home/klin/he/stackexp4.html
[]
```

## B. Sample XSL Code

The XML files produced by KUMO as described in Section 2.1.2 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="kumo.logic.fol.ConjunctionEli">
   <a href="{constant(conjel)}" target="back">Conjunction elimination</a>
   yields the following <xsl:apply-templates match="count"/> subgoals:
</xsl:template>
```

It 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`"; a link to the conjunction elimination tutorial page, the URL of which is named by the constant "`conjel`", is also inserted. (The list of subgoals is already present in the XML tree that is being processed and therefore does not need to be inserted by this rule.)

## C. A Formal Description of 2-Doags

Here we briefly define the data structure that is used for storing 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 $d_0\colon F \to N$ and $d_1\colon F \to N^*$ (called **source** and **target**), a set $W$ of "labels" plus a labeling function $l\colon N \to W$, 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 2.2.3. 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. A 2-occurrence in $D$ is just a path in $D^\flat$.