Web-based Multimedia Support for Distributed Cooperative Software Engineering

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

Abstract The Tatami project is building a system to support software engineering over the internet, exploiting recent advances in web technology, interface design, and specification. Our effort to improve the usability of such systems led us into algebraic semiotics, while our effort to develop better formal methods for distributed concurrent systems led us into hidden algebra. We discuss the Tatami system design, especially user interface issues, and sketch an extension of algebraic semiotics for interface dynamics.

1 Introduction

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

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

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

The second goal is motivated by the observation that large projects have multiple workers, often at multiple sites with different schedules. It is therefore difficult to share information, coordinate tasks, and maintain consistency. We seek to ameliorate this with generic tools to support distributed cooperative work over the internet. An unusual feature needed by our application to maintain the truth of proofs parts in the face of constant change, including insertion of new proof parts, distributed over both space and time; this is not supported by standard tools for version and configuration management. The advantage of a generic tool is its flexibility, including its potential use for other applications.

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

Formal methods have been used to prove the correctness of software systems, but this is known to be very difficult. New technologies such as the web, multimedia applets, Java, XML, XSL, and JSP offer opportunities to reduce this difficulty that have not yet been much explored. Since we wish to assist ordinary software engineers in using formal methods to design and verify complex systems, an important task for our project is to find better ways to explain and document proofs. Examining mathematics books and papers, even in logic, shows that mathematicians almost never write formal proofs in the strict sense of mathematical logic. The only proofs written this way are very simple illustrations of formal proof methods, not proofs of genuine mathematical interest. This

1This research was supported in part by National Science Foundation grant CCR-9901002, and by the CafeOBJ project of the Information Promotion Agency (IPA), Japan, as part of its Advanced Software Technology Program.
is because all but the simplest fully formal proofs are practically impossible to comprehend. Unfortunately, these are exactly the kind of proofs that are produced by mechanical theorem provers.

To improve this sad situation, we suggest making explicit the motivation and structure of proofs, and integrating them with relevant background and tutorial material. These recommendations are consistent with ideas from cognitive psychology, narratology and semiotics, as discussed further in Section 3. In particular, the structure of our proof websites was designed using algebraic semiotics [10, 11], which combines algebraic specification with social semiotics. Algebraic semiotics provides a way to formalize user interface designs and to compare them for quality, based on how well they represent what is important in the underlying functionality; although it uses formal methods, algebraic semiotics does so in a way that remains sensitive to social context; see Section 3.2.

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 and using tools. See Section 2.3.3.

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

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

This paper extends, updates and amalgamates work reported in [11, 13] and other papers. The latest information on the Tatami project can always be found at its URL, www.cs.ucsd.edu/groups/tatami.

2 Tatami System Design

This section sketches the Tatami system design, including:

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

2.1 Architectural Overview and Some Implementation Details

Figure 1 shows the Tatami system architecture. Its most important components are the Kumo website generator and proof assistant, the tatami database, one or more proof engine (especially some version of OBJ), the tatami

---

\(^2\)At the time of writing, this was not yet implemented.

\(^3\)This is a Japanese word for spider.
protocol, and a standard web browser. This is a modular architecture, designed to facilitate use of a variety of logics. 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. Our current prototype uses the BOBJ (for Behavioral OBJ) [12] proof engine, but our Barista\(^4\) proof server could be used to wrap other proof engines, such as CafeOBJ [4] and OBJ3 [18]. The tatami database, BOBJ and Duck are implemented using Java technology, including JavaCC for parsing. XML files generated by Kumo are passed to a processor that uses an XSL style file to generate the HTML pages actually seen by users; users can write their own XSL style files for a different display style if they wish.

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

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

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

2.2 Using Kumo

Kumo executes proof commands to generate a proof tree, and then generates a website that documents that proof. This can be done in two modes, called local and database; cooperative work is only supported in database mode, since local mode places output in local files rather than the tatami database. Users mainly interact with Kumo through two specialized languages, BOBJ for (behavioral) specification, and Duck, which has commands for both proof execution and proof display. A typical session goes as follows (see Figure 2):

1. Choose a project; then load a specification, or introduce a new specification.
2. Select an existing proof task, or introduce a new proof task.
3. Enter or edit a Duck program, and then execute it in local output mode; this produces or updates the corresponding proof website.

\(^4\)This is an Italian word for a person who serves espresso.
4. View the proof website on a browser.
5. Repeat from step 2.
6. When done, execute in database mode; every subtask of the selected task is entered into the tatami database, and can be viewed by all project members.

![Diagram](image)

**Figure 2. The Edit-Execute-Browse Cycle**

While this edit-execute-browse cycle (again see Figure 2) might seem old fashioned to some readers, empirical studies [22] and our own experience have found that the current fad for direct manipulation interfaces for theorem proving is counter productive for complex proofs, although it has value as a pedagogical aid for small proofs, e.g., for beginning students of formal logic. The main problem is that the data structures that underlie proofs of real interest are too large to navigate easily by direct manipulation, in part due to the difficulty of knowing where you are, because of the homogeneous structure of proofs. Another problem is that specifications evolve in real projects, and this should be effectively supported. Although these points are simple, they are important for the practical use of formal methods tools, and they are neglected by most current tools. This may be due to a Platonic bias on the part of their implementers, as well as many of their users. Such a bias tends to exclude complex issues arising from the social and cooperative nature of work, and to ignore the limited rationality and embodiment of real workers, not excluding mathematicians. Of course, this is not an argument against more sophisticated proof environments, e.g., parallel to the integrated proof environments (IDEs) used in programming.

### 2.3 Tatami Databases

Any practical implementation of formal methods must support bookkeeping for the complex dependencies among proof tasks, subtasks, and specifications, in the face of their mutual evolution, and the difficulties of coordinating work over a distributed group. The Tatami system does this using a distributed database, the coherence of which is maintained by a specialized protocol (see Section 2.6). Each worker has a local database, conceptually divided into the three components described in the three subsections below.

#### 2.3.1 The Project Database

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

#### 2.3.2 The Specification Database

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

\(^5\)At the time of writing, items 2. and 3. were not yet implemented.
1. Importation of one specification by another.
2. Equivalence and enrichment of specifications.
4. Evolution from a previous version.

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

2.3.3 The Proof Database

The Proof Database keeps track of the support given for tasks and subtasks, which can range from fully formal mechanical proofs to informal “back of envelope” arguments. Alternative validations of the same task are also allowed. Instead of a precise description of how these capabilities can be achieved (for which see [13]), here we stress their importance for a system intended to be used in a practical way for real projects.

Recall that fully formal proofs are not found in real mathematics, for the very good reason that they would be too difficult to comprehend, and too difficult to produce. The same reasons apply to formal proofs in computer science. Commercial software projects usually cannot afford the overhead that would arise, especially in view of the burden of updating proofs in the face of evolving requirements and the consequently evolving specifications and code. However, informal or semi-formal arguments for correctness are often developed, and it would surely be a good thing if these could be integrated into the documentation of the system as it is produced [8].

As for multiple proof attempts, these often arise for a difficult proof task, because it may be unclear which, if any, will succeed. Finally, we note that ordinary 2-valued logic is not suitable for this application, because the credibility of various informal arguments will in general lie somewhere between true and false. This suggests using a fuzzy logic for credibility. However, the usual fuzzy logic does not capture the multiplicative effect of uncertain arguments; for this, we use the variant of fuzzy logic introduced by Goguen in [7], which represents conjunction with multiplication and disjunction with the maximum operation.

2.4 Hidden Algebra and BOBJ

The Tatami project uses an approach to behavioral specification and verification called hidden algebra, which allows models that only “appear” to satisfy their specifications, in the sense of exhibiting the same behavior under all relevant experiments [9, 12]. This is important because many clever implementations used in practice only satisfy their specifications behaviorally in this sense. The most important method we have for proving behavioral properties, such as the behavioral correctness of an implementation, is coinduction, and we have recently implemented in BOBJ an especially powerful automatic form of coinduction called circular coinductive rewriting [12]. Hidden algebra can handle all the main features of modern software systems, including states, classes, subclasses, attributes, methods, abstract data types, concurrency, distribution, nondeterminism, generic modules, and even logical variables (as in logic programming) [16].

This approach regards code as secondary, because it can be (relatively!) quickly written or even automatically generated from specifications that are sufficiently modular and detailed; moreover, empirical studies show that little of software cost comes from errors in coding [2]. This implies a focus on specification and verification at the design level, avoiding the ugly complications of programming language semantics. The BOBJ [12] language extends the classical algebraic specification language OBJ [18] by adding support for hidden algebra; this is

\[\text{6The usual fuzzy logic represents conjunction with minimum and disjunction with maximum [28]; the more general approach of [7] introduced the notion of “complete lattice ordered semi-group,” a structure that has more recently become popular under the name “quantile,” and of which the logic of Kumo is a special case. This logic allows for the cumulative effect of uncertainty in reasoning chains, while taking the most promising choice when there are more than one.}\]
similar to the CafeOBJ language [4]. The Tatami system and Kumo further extend the logic by allowing first order sentences with behavioral equations as atoms, plus induction for initially defined data types.

### 2.5 The Duck Command Language

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

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

Some of these rules are automatic, in the sense that they are always applied, unless explicitly inhibited by the user, while other must be explicitly invoked; the latter include those rules that require the users to supply parameters.

Duck also has another sublanguage for proof display. This provides the information that Kumo uses to produce the XML file containing the tree structure of the proof\(^7\). For example, when conjunction elimination is applied to the goal \( T \vdash \Sigma A \land B \), the two new subgoals, \( T \vdash \Sigma A \) and \( T \vdash \Sigma B \), are added to the XML file. Kumo then uses an XSL file to generate HTML text for the proof; at this stage, links to tutorials, machine proof scores, and explanation are added, as described in more detail in Appendix A. Different XSL files will generate different displays. Duck also automatically provides appropriate cross-references to any other proofs that are used, such as lemmas.

### 2.6 The Tatami Protocol

A problem with using 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 unless \( p_2 \) has already been verified and entered. Inconsistencies can also arise when items are deleted. The tatami protocol maintains the consistency of the tatami databases, taking account of the following situations (see [17] for a detailed description):

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

We have used Kumo to formally prove the correctness of a version of the tatami protocol with respect to a communication medium that can lose or duplicate data [17], and we have implemented it using the IP internet protocol.

### 3 User Interface Design

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

\(^7\)However, if no information is provided by the user, then Kumo uses default values to produce reasonable output.
3.1 Narratology

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

An important resource for this is the theory of oral narratives developed by William Labov [19], who showed that these have a precise structure, which includes the following:

1. an optional orientation section, which provides basic orientation information, such as the time and place of the story;
2. a sequence of so-called narrative clauses which describe events;
3. it is assumed by default that the ordering of these clauses corresponds to the temporal ordering of events they describe — this is called the narrative presupposition;
4. the narrative clauses are interleaved with evaluative material which “evaluates” the events, in the sense of relating them to socially shared values;
5. finally there is an optional closing section, which may contains a “moral” or summary for the story.

The above follows [20, 21], which describe more recent developments than [19]. Aristotle [1] also gave some useful guidelines, including unity of time and place, having a beginning, middle and end, and the skillful use of language, especially metaphor.

The work of Joseph Campbell and Christopher Vogler [27, 3] on the role of characters in stories, especially the role of heroes, is also relevant; the dramatic importance of having the hero tested by obstacles is emphasized by these authors, and we have used it to structure some of our proof websites, since obstacles are exactly what is needed for creating drama. Some experimental results from [14] on multimedia instruction are also relevant, suggesting that narrative, especially when it is oral (i.e., in the audio medium), is important for controlling the interpretation of material in other media. It is also well known in cinema that exactly the same scene with a different narrative can have a completely different interpretation and emotional effect; music can also have a powerful effect on interpretation. Some influences of narratology on our proof website design conventions are discussed in Section 3.3.

3.2 Algebraic Semiotics

A basic insight of Ferdinand de Saussure [25] 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 [15], as a loose algebraic theory (consisting of a signature and some axioms) plus some further structure specific to semiotics, including constructors, a hierarchy of levels for signs, and priorities among constructors [10]; a related insight is that representations can be seen as translations or maps from one sign system to another [10]; it is similarly natural to formalize these translations using the algebraic specification notion of theory morphism. Our case studies show that maps between sign systems in general do not fully preserve structure, and in particular, must involve partial functions. These considerations motivate the definition of semiotic morphism given in [10]; a key point is that the quality of a representation can be examined in terms of what is preserved by its semiotic morphism.

User interfaces are of course prime examples of representations, and it is natural to study them using semiotic morphisms which map from the sign system of the underlying functionality to a sign system for displays; the
quality of the interface is then measured by the quality of its semiotic morphism. Details, some of which are quite technical, are omitted here, but may be found in [11] and [10]. In particular, note that this theory provides a natural way to handle multimedia displays.

3.3 Proofwebs and the Tatami Conventions

Professional advice for user interface design in general, and for website design in particular, nearly always calls for using style guidelines to produce a uniform “look and feel” that is appropriate for the particular application, e.g., see [24, 26]. We have developed the following tatami conventions (updated from [11, 13]) as style guidelines for proof websites generated by Kumo. To clarify the discussion, we distinguish sign systems for abstract proofwebs and display proofwebs; the first contains the proof information, while the second also includes display information. We will use the word “unit” for any block of information of the same kind in a display proofweb.

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

2. **Tatami pages** are the basic constituents of display proofwebs; each tatami page has one or more proof units showing its inference rule applications, interleaved with one or more explanation units; it is feasible to have both on the same web page because there should only be a small number of proof steps per page (about 7 non-automatic rules works well).

3. The explanation units of tatami pages are prover-supplied informal discussions of proof concepts, strategies, obstacles, etc.; these may contain graphics, applets, and of course text.

4. Tatami pages can be browsed in an order designed by the prover to be helpful and interesting to the reader; if possible, these pages should tell a story about how obstacles were overcome (or still remain); this order will be called the narrative order.

5. Major proof parts, including lemmas, have their own subwebsites, each with the same structure as the main proof, including homepage and explanation units. These appear in a separate dedicated persistent popup window.

6. Tatami pages also have associated formal proof scores, which appear in another separate popup window when summoned from a tatami page. It is convenient to have a separate window because users usually want to look at the proof and explanation at the same time as the proof score. Readers can also request proof score execution, and the result is displayed in the same window as the score, so that one can easily alternate between them.

7. Major proof parts can have an optional closing page, to sum up important results and lessons; these appear in the same window as their tatami pages.

8. Formal proof steps are automatically linked to pre-existing tutorial background pages; e.g., each application of induction is linked to a webpage that explains the kind of induction used. Tutorial pages have their own dedicated persistent popup window.

9. A menu of open subgoals appears on each homepage, and error messages are placed on appropriate pages. In database mode, a summary of this information may be seen in the status window, which is a specialized popup that reports any currently open subgoals.

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

3.4 Justifying the Design Guidelines

Having homepages for major proof parts is motivated by the “orientation” sections in Labov’s theory of story structure [19]; homepages appear in the same window as their tatami pages, since they are part of the same
narrative flow. The closing webpage is justified the same way, though these are less common.

Interleaving prover-supplied informal explanation pages with proof steps was suggested by the similar interleaving of narrative and evaluative material in stories (see Section 3.1 and [19]); the evaluative material provides the motivation for important proof steps, by relating them to values shared among the appropriate community of provers. Limiting the number of steps to approximately 7 is justified by classic work of Miller [23].

The remaining design guidelines are justified using algebraic semiotics. The basis for these arguments is that any display to users can be seen as a semiotic morphism from the sign system for abstract proofwebs to that of display proofwebs, and that these morphisms are compared for quality, based on how well they preserve source structure and content.

- **Windows:** The main contents of a display proofweb are its proof steps, informal explanations, tutorials, and mechanical proof scores. These four are also the main contents of abstract proofwebs, and their preservation determines the quality of their representation. The basic classification into four sorts is reflected in our choice of windows for displaying them. Because tatami pages are the main constituent of proofwebs, theirs is the master window, and because explanation pages are so closely linked, they share that window; each unit is enclosed in its own “box.” Tutorial and machine proof score pages each have a separate window.

Rewrite above: discuss as an ADT with four main sorts.

- **Backgrounds:** Each sort of unit has its own background color: proof units have light beige, explanations have light yellow, tutorials have yellow marble, and proof scores have light purple. Although the choice of colors is somewhat arbitrary, and is easily changed by editing the XSL file, their distinctness reflects the importance of distinguishing these four units.

- **Navigation:** Similar considerations hold for navigation. Each page has a title, supplied by the user in the Duck script. Buttons are used to move to other pages of the same sort, and to open windows that display information of other sorts. Each persistent window has somewhat different layout and navigation buttons, reflecting its different use. For example, the master tatami window has “NEXT” and “PREV” buttons to step through the narrative ordering of tatami pages, as well as buttons that support conventional tree traversal (“LEFT”, “RIGHT”, etc.).

- **Mathematical Formulae:** We use .gif files for mathematical symbols, in a distinctive blue color, because mathematical signs come from a domain that is quite distinct from that of natural language. (We hope to use MML when it becomes available.)

Some additional applications of semiotic morphisms to the user interface design of the Tatami system are described in [11], in a more precise style than here, although they are based on an older version of the system. For example, [11] shows that certain early designs for the status window were incorrect because the corresponding semiotic morphisms failed to preserve certain key constructors. The graduate user interface design course at UCSD uses algebraic semiotics (see www.cs.ucsd.edu/users/goguen/courses/271), and more information can be found there.

### 3.5 Dynamic Algebraic Semiotics

This section sketches a new generalization of algebraic semiotics to handle dynamic interfaces by extending its foundation from classical algebra to hidden algebra. As a simple example, consider the problem of designing that part of the user interface that supports browsing the proof's produced by Kumo. This must support traversal of the proof author’s narrative order using the operations “NEXT” and “PREV”, as well as traversal with the tree oriented operations “LEFT”, “RIGHT”, etc. It is common practice to draw an automaton having one state for each proof tree node, and a transition label for each traversal operation. But this does not allow for the fact that each proof has a different structure, and thus a different automaton, nor does it account for the display produced in each state, nor for the variety of possible implementations of transition lookup, e.g., using lists, arrays, or hash tables. In fact,
this practice is purely intuitive, and provides no basis for the rigorous mathematical analysis of possible designs. In order to address the display, implementation and quality questions raised above, the automaton model would have to be supplemented in various ad hoc ways, whereas our approach can handle all of these in a single unified framework.

There is insufficient room here for details, but we can say that hidden algebra, with its distinction between sorts used for states and for attribute values [12], provides a precise way to represent situations like that described above; moreover, the corresponding extension of semiotic morphisms then gives a precise basis for comparing the quality of interface designs realizing the desired dynamics, without bias towards any particular implementation.

4 Conclusions and Future Research

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

Topics that would be interesting to explore in future research include: making proof displays more interactive; evaluating the use of audio, avatars, chatrooms, and archetypal characters; use of a fuzzy critical path algorithm to choose the most helpful open subgoal for further development; further developing the theory of hidden semiotic morphisms for dynamic displays; and integration with Gibson’s notion of affordance [5, 6]. In addition, we hope to use the Tatami system in teaching the UCSD graduate course on programming languages, which would stimulate further developments to the system, its interfaces, and its theory. Finally, we feel we are now in position to begin developing methodological guidelines for applying hidden algebra and the Tatami system to difficult applications like communication protocols.

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

References


A Sample XSL Code

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

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

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