Social and Semiotic Analyses for Theorem Prover User Interface Design[1]

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

Abstract: We describe an approach to user interface design based on ideas from social science, narratology (the theory of stories), cognitive science, and a new area called algebraic semiotics. Social analysis helps to identify certain roles for users with their associated requirements, and suggests ways to make proofs more understandable, while algebraic semiotics, which combines semiotics with algebraic specification, provides rigorous theories for interface functionality and for a certain technical notion of quality. We apply these techniques to designing user interfaces for a distributed cooperative theorem proving system, whose main component is a website generation and proof assistance tool called Kumo. This interface integrates formal proving, proof browsing, animation, informal explanation, and online background tutorials, drawing on a richer than usual notion of proof. Experience with using the interface is reported, and some conclusions are drawn.

1. Introduction

Recent large advances in performance have made it arguable that the most pressing open problems in theorem proving today concern user interface design. The complexity of non-trivial formal proofs makes it difficult for users to navigate them, and even more difficult to understand, improve or debug them. The range of specific issues is wide. Resolution-based provers by their nature yield proofs that are unstructured and low level. Fully automatic provers are difficult to control, and often yield proofs having counter-intuitive structure. Proof checker commands are often low level, and can be tedious to use, read, and write; higher order tactics certainly improve this situation, but can still be difficult to read, write, and debug. In addition, theorem prover command languages generally lack the sophisticated structuring features found in modern programming languages. As a result, even well trained and highly motivated users often become frustrated. We believe that better interface design, which we take to include structure supporting proof command languages, can go a long way towards reducing this frustration, improving productivity, and helping users who are not experts in formal methods.

We advocate an approach between the two extremes of fully automatic theorem proving and low level proof checking, suggesting that users should do the high level proof planning, while machines do low level computations such as term rewriting and resolution. For this to be effective, users need precise feedback on how proof subgoals failed, not just some (potentially huge) unstructured proof tree. In fact, since nearly all of a user's effort during proving goes into proof

[1] This research was supported in part by the CafeOBJ project, under management of the Information Technology Promotion Agency (IPA), Japan, as part of its Advanced Software Technology Program.

debugging, it is essential to support the cycle of proof-attempt, failure-diagnosis, and proof-retry, in the most effective way possible. This should in particular include powerful structuring mechanisms for both the input and the output of a mechanical prover; our input language is called "Duck," and our outputs are interactive hypermedia websites.

Technology-driven approaches to user interface design (hereafter, UID) are common, and graphical user interfaces are popular. But recent work in requirements engineering and human-computer interaction (hereafter, HCI) emphasizes the importance of identifying specific user roles and their associated requirements, and then focusing design on those requirements, as opposed to using whatever technology has recently become possible and/or fashionable. For example, recent HCI research shows that graphics is not always better than text, and that certain multimedia combinations often work better than any single medium [46]. In particular, a clickable proof graph can be difficult to use for all but the smallest proofs[2], and many serious users report that for this reason, they prefer command line interfaces over direct manipulation graphical interfaces [41]. Methods from ergonomics and experimental psychology are time-consuming and expensive, and moreover do not easily address the important issues of structure, interaction, and user requirements.

General principles from UID and HCI are useful, but cannot take sufficient account of the specific requirements of theorem prover users. Work on the nature of proving like that in [40] helps fill this gap, as do social analyses of actual verification efforts such as [17]. This paper approaches user interface design using principles from requirements engineering, HCI, CSCW (computer supported cooperative work), cognitive science, social science, narratology, and especially semiotics, which is the study of signs. General references on semiotics include [45, 52, 15, 35]. Semiotics has generally not been developed in a mathematically rigorous way, has not systematically addressed issues of quality, and has not taken sufficient account of social context. We try to address these limitations with a new approach called **algebraic semiotics** [19], which combines algebraic specification [25] with social semiotics in the sense of [18].

This paper considers some user interface design issues for the **Kumo**[3] [22] website generator and proof assistant, which is the most important component of the tatami[4] system. Although we have found that well designed proof websites can make it easier to understand proofs, experience has also shown that websites (like ours) having complex interaction (buttons, popups, applets, etc.) and complex links are difficult to build and even more difficult to maintain. Page-oriented commercial products like Microsoft's FrontPage do not address cross-page issues such as link maintenance and site-wide design conventions; in fact, FrontPage makes such issues more difficult to address in some ways[5]. A website generator can greatly reduce the effort involved, doing many routine tasks by filling slots in predefined templates, e.g., with web links. However, Kumo does much more

---

[2] I first encountered this using the Jape and 2OBJ systems at Oxford. The difficulty is knowing where you are in a large tree, all nodes of which tend to look similar. However, such systems can be useful for pedagogical purposes.

[3] Kumo is a Japanese word for spider.

[4] "Tatami" are natural fiber mats used in traditional Japanese homes, about 5 by 3 feet in size. A 2 tatami room, like a 2 tatami proof, is pretty small, but a 20 tatami room (or proof) is large and should probably be subdivided.

[5] Version 1.1 from late 1997 was evaluated; later versions may be better.

than this, since it also helps with proof planning, generates proof scripts that are sent to proof servers for routine proof subtasks, helps with debugging proofs, and coordinates cooperative work; see Section 2.

Our approach to user interface design considers social aspects important for determining requirements. The social analysis used here is a pragmatic blend of methods from discourse analysis [38, 37], narratology (which is the theory of stories), and ethnomethodology [12, 51], which focuses on the concepts and methods that members themselves actually use. However, this paper emphasizes the results of such analyses and their application, rather than the technicalities of how conclusions are obtained; for more detail on that topic, see [24, 16, 18]. Algebraic semiotics provides a precise way to describe a user interface as a "semiotic morphism" from one sign system to another. As discussed in Section 3.1, the basic intuition is that the source of the morphism is the underlying system, its target is the user interface, and its structure gives a precise description of how the functionality of the system is accessed through the interface. Algebraic semiotics also provides certain precise notions of quality for morphisms, which are applicable to user interfaces. Our intention is not to generate user interfaces from these mathematical descriptions, but rather to use the descriptions as models, to check that certain desirable properties hold, much as an electrical engineer can use a mathematical model of a circuit to calculate (for example) the current through a resistor, to check that it is not so large as to damage that component.

There are large literatures on UID, HCI, CSCW, theorem proving, semiotics, and formal methods; all of these are relevant, but to keep the paper short, we have only given citations to work that directly influenced our own. We are not aware of prior work that applies semiotics or ethnomethodology to designing proof representations, that views proofs as more than just graphs of inference steps, or that considers distributed cooperative proving.

1.1. Three Roles in Theorem Proving

Guidelines for user interface design (e.g., [53, 1]) usually emphasize the importance of determining who your users are, and what their requirements are, noting that there may be more than one distinct kind of user, and hence more than one distinct set of requirements. We distinguish three different roles[6] that occur in theorem proving: (1) specifier; (2) prover; and (3) reader. The specifier writes the theory in which the proof will be undertaken, and states the conjecture to (perhaps) be proved; the prover tries to actually carry out that proof task; and the reader tries to understand the specification, the conjecture, and especially the proof. Usually provers are also proof readers, because (unless their proof works perfectly the first time – which is unlikely), they need to understand what parts of it succeeded, what parts failed, and why, in order to make further progress. Moreover, provers are also often specifiers.

These three roles have different requirements, which can very well conflict. Provers want proof scripts to be as easy to write, edit, and run as possible, hiding as much detail as possible, and (in general) they prefer to use shortcuts and abbreviations. However, proof readers value understandability, (often) want

[6] We speak of **roles** rather than user classes, because social classes are notoriously difficult to pin down, and in any case, several different roles can be filled by the same person.

to see motivation and explanation, and may be hindered by shortcuts and abbreviations. Meanwhile, specifiers want to express the basic concepts involved in as clear and concise a way as possible. This analysis implies that each role should have a different interface (which may be just a text-based language) to support its own requirements. This paper focuses on interface design for the reader, where we try to determine how to best exploit the structured browsing and display capabilities of web technology. Subsequent papers will focus on the other two roles, e.g., [21] considers specification language design; however, we do briefly discuss the current status of these interfaces in this paper.

The technical justification for distinguishing these three roles proceeds as follows: we first define a *role* to be a significantly frequent cluster of coherent user scenarios, where user goals are considered to be part of the notion of scenario. Such clusters are identified through the usual techniques of analysis of transcripts, interviews, participant observation, etc. (e.g., see [24]); two important datasets for this purpose are described in [17]. The three roles then emerge as three major clusters of scenarios (or more technically still, coherent segments within scenarios).

2. Overview of the System

Interface design should take account of the nature of the system to which the interface provides access. Therefore we begin with an overview of our system; not everything described here has yet been fully implemented, and many details are omitted. See `www.cs.ucsd.edu/groups/tatami` for the latest information.

The purpose of this system is to support teams in the design, specification, and validation of software (and/or hardware) systems, especially distributed concurrent object systems. Because of this, it differs from related systems with which we are familiar in several respects: (1) web-based interactive documentation is automatically generated; (2) design is separated from validation, with distinct languages for each activity; (3) a range of formality is supported, from full mechanical proofs to informal “back of envelope” arguments, using a fuzzy logic for confidence levels; (4) distributed cooperative work is supported; (5) there are links to online tutorials on background material; (6) there is a distributed database; (7) a specialized protocol maintains logical consistency in the presence of semi-reliable internet communications; and (8) design and validation of concurrent object systems are supported through novel facilities for behavioral specification and coinduction, based on hidden algebra.

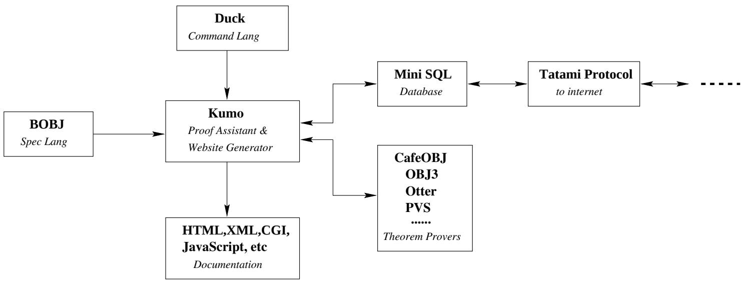

Fig. 1. System Architecture

Figure 1 shows the current system architecture. The most important component is the Kumo website generator and proof assistant; other components include one or more proof engine (especially some version of OBJ), the database, the protocol, the barista proof engine server, and a standard web browser. Kumo assists multiple distributed users with design and validation, executes Duck proof commands over specifications written in the BOBJ language (see Section 2.1), sends proof scripts out for (possibly remote) execution, and generates documentation websites that follow the "tatami" design conventions (see Section 3.4). Multiple versions and incomplete activities are supported at multiple sites, and logical consistency of the databases is maintained by the tatami protocol. The database is a MiniSQL server [36] written in Java.

Several websites have been generated by Kumo, of course using files designated by users for specifications, goals, and explanations; links are automatically generated to tutorial webpages for topics including first order logic, proof planning, hidden algebra, and coinduction. These examples are organized to form a gradual introduction to the methods, tools and processes of the project:

1. An inductive proof that $1 + ... + n = n(n + 1)/2$.
2. A coinductive proof of a behavioral property of a simple flag object.
3. A proof of behavioral correctness of the array-with-pointer implementation of stack, using coinduction (see Appendix A).
4. An inductive proof that the reverse of the reverse of a list is the original list, with several lemmas, including that append is associative.
5. A coinductive behavioral refinement proof of the correctness of implementing sets with lists.
6. An inductive proof of a formula for the sum of the squares of the first n natural numbers.
7. A first order logic proof that the square root of 2 is irrational.
8. A coinductive correctness proof for the tatami protocol.

These can be seen at `www.cs.ucsd.edu/groups/tatami`; the last proof is the most sophisticated. Each proof is guaranteed formally correct, because Kumo generated each inference step, and checked it on a proof engine. The homepage of each example gives further information, usually including an interactive applet to illustrate important intuitions about the problem and proof[7]. The website also has an industrial style correctness proof for an optimizing compiler for OBJ, following [34]; this proof was not done by Kumo, and is partially informal[8].

Web technologies provide powerful and rapidly evolving support for asynchronous distributed, multimedia systems, and it is doubtful that we could have built such a complex system so quickly using the technologies of even a few years ago; in particular, we have used Javacc, XML, and dynamic HTML.

[7] Of course, not all proofs admit a picture or an applet to illustrate their main ideas, or even their result, but when they do, readers can explore issues for themselves, and thus alleviate the pain of purely passive reading; interactive browsing and the background tutorials also help.
[8] Thanks to Dr. Akira Mori.

2.1. BOBJ

BOBJ (for Behavioral OBJ) can be considered a variant dialect and extension of the CafeOBJ language [7, 6]. Like CafeOBJ, it extends the functional capabilities OBJ3 [10, 32] with behavioral modules, for the behavioral specification of objects (i.e., state machines); in addition, it provides modules for first order sentences with loose semantics, a capability that CafeOBJ currently lacks. The underlying logic is first order logic with equations as atoms, and there are three kinds of module, for defining data, objects, and properties, each with its appropriate interpretation for equality, which are initial algebra semantics for data, hidden semantics for objects, and both for first order theories, in each case supporting subtypes through order sorted algebra [28]. First order sentences can occur only in modules having loose semantics. As in CafeOBJ, behavioral modules support the notion of coherence, as introduced by Diaconescu [5], but in a generalized form [56] called behavioral congruence, that allows more than one hidden argument in behavioral operations. Type checking and execution for BOBJ are implemented in Kumo by translation to OBJ; there is no separate interpreter.

Behavioral specification and verification are more important than seems to be generally realized. For a simple example, a stack specification can be refined to a composition of atomic specs for array and pointer, or more precisely, an enrichment of the sum of atomic specs for array and cell. This does *not* strictly satisfy the stack spec, but it does *behaviorally* satisfy it; thus behavioral refinement is needed here. Software implementations often require behavioral satisfaction for their correctness, due to clever optimizations of space and/or time, such as sharing storage among subtasks.

BOBJ has a sophisticated module system based on ideas from parameterized programming [14, 31], using module expressions to describe how to put modules together to form systems, with a rich set of composition operations. Multiple inheritance is supported at the module level. We hope eventually to add higher order modules and views to BOBJ, along the lines of [27], to enhance support for specification reuse and facilitate applications to areas like software architecture.

Following Section 1.1, theorem proving commands are *not* part of BOBJ, but instead appear in the separate Duck command language. This separation of verification from specification allows each language to be better adapted to the needs of its corresponding role; for example, BOBJ is declarative, whereas Duck is imperative. Again following Section 1.1, the generated proof websites are in yet another "language" (first XML, which is translated to HTML, JavaScript, etc., and eventually displayed as graphics). This three-fold language separation also promotes specification reuse, and makes it easier to improve the system, because of increased modularity.

This paper is not an appropriate place for details about the semantics of BOBJ, but briefly, its models are algebras[9] having a fixed model of the data as a subalgebra, and satisfying the relevant sentences, with equality interpreted initially, behaviorally, or loosely, as appropriate. Further information is given in Section 2.4; see [26] for a detailed introduction to hidden algebra, [21] for more on BOBJ, and [56] for recent work on coinduction and behavioral congruence.

[9] Code written in a programming language can be regarded as such a model.

2.2. Kumo and Duck

Kumo does two very different things: it assists with proofs; and it generates documentation. The proof websites that Kumo generates have complex structure, involving mathematical formulae (currently implemented using gif files) and user interaction through frames, hyper-links, applets, buttons, popups, etc. Kumo greatly reduces the effort of building and (more importantly) maintaining such websites, by automatically filling in predefined templates with links to tutorial webpages, machine proof code, etc. Kumo is site-oriented, rather than page-oriented like most HTML editors, and its output embodies our site-wide style conventions (see Section 3.4). Kumo is implemented in Java, by Kai Lin.

Kumo executes commands in the Duck language. A typical Kumo session involves iterative debugging and execution, with a browser open to view the generated website, and an editor open to update the Duck proof script and/or the specification:

1. Choose some existing unsolved task, or else a new task.
2. Enter and/or edit commands in the file, e.g., to select subtasks and execute rules. The latter may produce new subgoals, which can be named for later reference. Other commands control webpage format, e.g., where to begin a new proof page, and which proof page is to be next on the "narrative" tour of the proof (in the sense of Section 3.4). If the display section is incomplete, or even empty, then a set of default conventions is used to produce a reasonable display in any case.
3. When editing is complete, request execution by Kumo, which will update the database, build and/or update webpages, which will include error messages, if there are any. If a task is atomic (e.g., a ground equation), it is checked on a proof server, and the results are used in updating subgoals.
4. View the result on a browser, especially the error messages.
5. Repeat from step 2.
6. When done, submit the work to other sites interested in the same task; then everything below the selected task is considered fixed and is broadcast over the internet (see Section 2.3.2).

The theorem proving aspect of Duck is relatively conventional. There are basic commands for the usual proof rules, which are most often used in a top-down problem reduction mode. However the lemma introduction rule allows doing proofs in other orders. Sophisticated forms of induction are supported, as is coinduction [26], and some higher level tactics, for example, multiple conjunction elimination, which given $H \models_\Sigma A \wedge B \wedge C$, updates the underlying data structures with new subtasks for proving A, B and C from H.

This approach of reducing tasks to tractable subtasks that can be handed off to proof servers lets users do the high level proof planning while machines do the most routine work; in our experiments so far, about 20 to 200 times as many inferences are done by machines as by users, including many non-equational rules applied automatically; this ratio will improve as we learn more. We now use the OBJ term reduction engine, but our paradigm allows using servers for any combination of proof engines anywhere on the internet; for example, a Presburger arithmetic server would be useful. We do not wish to compete with powerful theorem provers that have decades of development behind them; instead, we wish to reuse them.

The most unusual aspect of Duck is its website generation capability. Kumo generates code in HTML, JavaScript, etc. following the tatami conventions described in Section 3.4, with user-supplied declarations for page headers, file placement, and explanation texts. In addition, because Kumo uses XML as an intermediate stage in generating output, it is easy to generate documentation in other forms, such as LaTeX, and thence postscript. A sample Duck script is shown in Appendix A. Note that Duck scripts are quite different from the OBJ (or other proof engine) scripts that are generated from them; for example, compare the OBJ3 proof script in Figure 2 with the Duck command script in Appendix A.

2.3. Distributed Cooperative Proving

Because we aim to help software engineers, it is helpful to review some aspects of software engineering practice (e.g., see [54]). Industrial software projects usually involve multiple workers, often at different sites and/or with different schedules; hence they are cooperative and distributed. Teams of hundreds are common, and thousands are not unusual. Documentation is often hard to find, out of date, incomplete, or non-existent; furthermore, requirements, specifications, and personnel are typically all changing. Thus it can be hard to share information and coordinate tasks. This motivates our goal of developing tools to support distributed cooperative work on tasks that include specification, verification, refinement, coding, and documentation. Since we wish to make formal methods more available to ordinary software engineers, an important 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.

2.3.1. Proof Servers

Proof servers are used in two different ways. First, when Kumo produces a proof subtask that can be handled, e.g., by term rewriting, it sends that task to an appropriate proof engine; the result is then used in further proof planning. Second, the proof scripts that are sent to proof servers are placed by Kumo on special webpages, where if they wish, proof readers can re-dispatch them to a proof server with the EXECUTE button, and then view the newly recomputed result. In each case, a (Java) client applet is downloaded from the server (for security, the client must come from the server through a socket); the proof script URL is passed to the server; the server fetches the proof script, executes it, and sends the result back to the client applet, which then displays it. Figure 2 shows a typical OBJ3 proof script window, which is part of the website for the proof that a pointer and an array behaviorally implement a stack. Notice its EXECUTE button on the bottom. Our generic Java proof server is called "barista" (Italian for a person who serves coffee), and was implemented by Dr. Akira Mori.

2.3.2. Distributed Truth Maintenance Protocol

Distributed cooperative proving over the internet, allowing multiple proofs for (sub)goals, requires keeping careful track of proof tasks and specifications, with their ownership and current status. This information is stored in an underlying ProofDag data structure (see Section 3.2) in the local database of each site.

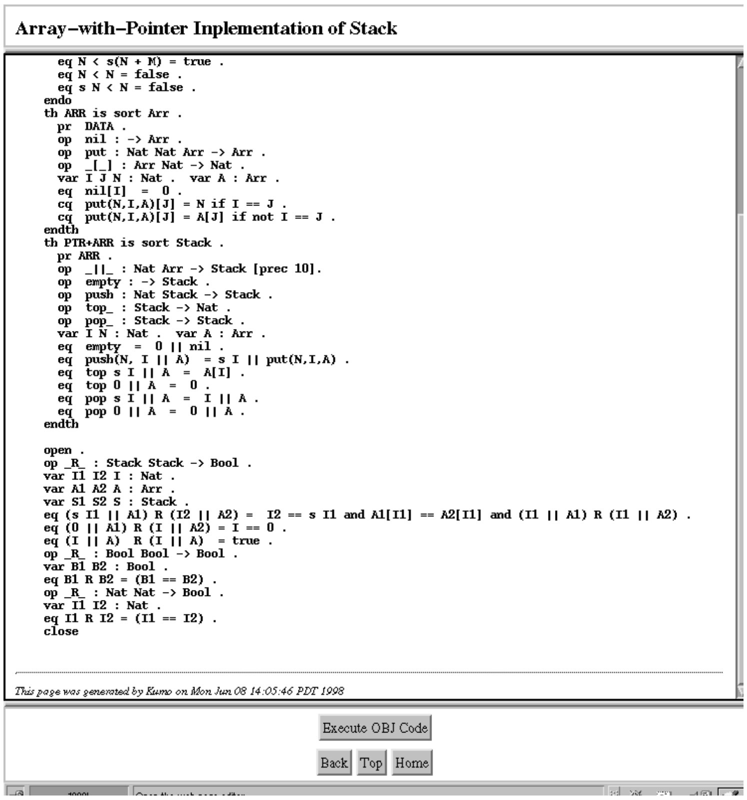

Fig. 2. An OBJ3 Proof Script Window

Whenever a user "registers" a proof fragment, this data structure is appropriately updated in the local database of every concerned site. Before registration, a proof fragment is only available in the user's local database; otherwise there could be an overwhelming profusion of proof parts that are no longer relevant. Users can delete their own proof fragments globally, but if some deleted subproof is used by someone else, then its ownership is reassigned to whoever used it first. All this must work correctly in the face of communications that are not entirely reliable. These considerations constitute some of the requirements for the tatami protocol, which is described in more detail in [22].

2.4. Underlying Logic

Because this paper is focused on user interface design, technical details about the logic that underlies the system would be out of place; nevertheless, some appreciation of how this logic impacts the work of specifiers, provers and readers will be helpful. A **proof task** has the form $H \models_\Sigma G$ with H a specification and G a **proof goal**; the symbol "$\models$" here does not indicate model theoretic satisfaction, but rather it separates the hypotheses from the goal; as is appropriate for a proof planning paradigm, the logic is a sequent calculus (see [20] for more detail). Inference rules reduce proof tasks to other proof tasks, and include the following:

- elimination rules for $\forall$, $\exists$, $\wedge$, $\vee$, and $\Rightarrow$;
- lemma introduction, case analysis, substitution and proof by contradiction;
- term rewriting and equational deduction;
- induction for data types (natural numbers, lists, etc.); and
- coinduction for behavioral properties.

This approach yields proofs that are (relatively!) simple and mechanizable, and (more importantly for the purposes of this paper) that leave the user in charge of the plan for the proof. The basic logic is (hidden order sorted) first order logic with behavioral and ordinary equality; both ordinary first order and equational logic are special cases. Although this logic has no predicates as such, this entails no loss of generality, because boolean valued functions provide the same power.

Hidden logic is concerned with behavioral properties; thus hidden specifications characterize how objects (and systems) behave, rather than how they are implemented. Hidden algebra is able to handle all of the typical features of the object paradigm, including classes, subclasses (inheritance), attributes, methods, and local state, as well as concurrency, distribution, nondeterminism, plus logical variables (as in logic programming), abstract data types, and generic modules [26]. Hidden algebra generalizes the process algebra, transition system and coalgebra approaches, in that methods and attributes can have one or more parameter. Coinduction is a new proof technique for behavioral properties. Sorts are used two ways in hidden logic: for data values (e.g., of attributes), and for states. These are dual: induction establishes properties of data types while coinduction establishes properties of objects with state. Similarly, initiality is important for data types, while finality is important for states. However, implementations need not be initial or final; this is significant because the best implementations often lie between these extremes. Hidden algebra is a natural next step in the evolution of the algebraic specification tradition that began with the initial algebra approach to abstract data types (ADTs) of [29, 30, 25].

2.5. Some Further Details

Although full formal verification is an option, we believe the most practical use of the system will often exploit the task structure of formal methods without the burden of completely formal proofs. This will ensure that all relevant dependencies are known, both in the logical sense of inferences, and in the process sense of relations among tasks; it will also ensure that we always know exactly where to put and where to look for documentation, test cases, etc. To support work that is informal, and hence not fully reliable, we associate confidence values in the unit interval with proof tasks, instead of Boolean truth values; this also allows techniques like critical path analysis to aid with task allocation. The fuzzy logic described in [13] is appropriate for computing these values, because it uses product to evaluate conjunction, instead of the classical minimum of Zadeh [61]; this means that the weakness of each constituent of a proof is reflected in the truth value of its conclusion; however, both fuzzy logics use maximum for disjunction, which returns the truth value of the best disjunct.

Requirements and specifications are key entities in a software engineering project database; code is less important, since it can be written (relatively) quickly, or even generated from specifications that are sufficiently modular and detailed. The most important relations among requirements and specifications are refinement and composition. A refinement relation says that one specification gives a way to realize the behavior of another. Relations of composition are described using the module expressions of parameterized programming [14]. Validations, which may or may not be formal proofs, are stored in the same database.

2.6. Some Related Tools

We briefly discuss a small number of particularly closely related tools. CafeOBJ [11, 7] is a Japanese project to build an industrial strength version of the OBJ language. The CafeOBJ language provides many interesting novel and powerful features that support behavioral specification and reasoning, based on hidden algebra; it also supports rewriting logic, but it does not provide first order proof assistance, documentation generation, or database capabilities. The CASL system [4], being developed by a diverse European group called CoFI (for Common Framework Initiative), aims to be a Common Algebraic Specification Language, but lacks features to support hidden algebra, order sorted algebra, and symbolic execution by term rewriting. Maude [3] extends OBJ3 with powerful features for rewriting logic and reflection. All three languages use parameterized programming [14] as the basis for their module systems.

The SpecWare system [55] from Kestrel generates code from detailed specs, using optimizing transformations to improve the quality of this code. SpecWare also has a module system based on parameterized programming, and in particular has `colimit` as a top level command. The LILEANNA system [58, 57] fully implements parameterized programming, and builds systems by composing compiled Ada modules. SpecWare has a verification capability, whereas LILEANNA does not. Prof. Peter Padawitz has developed a theory called "swinging types" that includes coinduction principles, and has an associated verification system [43, 44]. Larch [33] has a theorem pover for algebraic specifications, that has influenced some aspects of the Duck design (see Appendix A). None of the systems mentioned in this section support distributed cooperative work, or give any special consideration to user interface design or the readability of proofs.

3. Design Considerations

This section introduces algebraic semiotics, briefly discusses narratology and cinema, and describes our style conventions for proof websites. Then we give two ADTs that underlie the theorem prover, showing how our status window relates to one of these ADTs via a semiotic morphism. Finally, we apply our theoretical machinery to justify a number of basic user interface design decisions for Kumo.

3.1. Algebraic Semiotics

The most novel technique that we use for designing interfaces is algebraic semiotics. This paper is not the right place for a comprehensive exposition, as some aspects are quite technical, and even use a little category theory. Despite the mathematical formalism of the definitions below, the main ideas are very intuitive. For more detail, see [19], which is the best exposition now available.

ProofWebs, proof pages, and status windows are all classes of complex signs[10]. An important insight attributed to Ferdinand Saussure [52], is that signs should not be considered in isolation, but rather as elements of *systems* of related signs, including their structural aspects. Thus it is natural to think of a sign system as a set of signs, grouped into sorts and levels, with "constructor" functions at each level that build new signs from old ones. But such a set-based approach does not capture what I call the "openness" of sign systems, which includes the possibility that there might be other signs in the system that we don't yet know about, or haven't wanted to include; in fact, users of natural sign systems are always constructing *partial* understandings out of their experience so far. For these reasons, it is better to view sign systems as *theories* than as pre-given set theoretic objects. This motivates the following:

Definition 3.1. A **sign system** S consists of:

1. a set S of **sorts** for signs[11];
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, and there is a unique sort of maximal level, called the **top sort**;
5. a set C_n of level n **constructors**, used to build level n signs from other signs at levels n or less, and written $r\colon\ s_1 ... s_k d_1 ... d_l \to s$, indicating 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 sentences (in the sense of logic), called **axioms** that constrain possible signs.

□

A sign system is a loose algebraic theory with further structure (including level and priority); hence we can use algebraic specification techniques [25], e.g., in Section 3.5 to partially specify proof page and status window sign systems.

We illustrate the parts of this definition with a very simple *time of day* sign system. It has just one sort, namely time, and two constructors, one the constant time 0 (for midnight), and the other a successor operation s, where for a time t, $s(t)$ is the next minute. There are no subsorts, data sorts, levels, or priorities. But there is one important axiom,

$$s^{1440}(t) = t\ ,$$

[10] Readers with a philosophical inclination should note that we do not presuppose a "realist" view of signs as actually existing "real" entities, but instead, we consider sign systems to be models that are constructed for some pragmatic purpose, which in this paper is user interface design for a theorem prover. See [18] for further philosophical discussion.

[11] The denotations of sorts in models are not necessarily disjoint from each other.

where s^{1440} indicates 1440 applications of s, or more prosaically[12],

$$s^{1440}(0) = 0 \ .$$

These axioms capture the cyclic nature of time over a day; any reasonable representation for time of day must satisfy this condition.

The purpose of semiotic morphisms[13] is to provide a way to describe the movement (mapping, translation, interpretation, representation) of signs in one system to signs in another. For example, proof pages and status windows result from translating ProofWebs. Just as we defined sign systems as theories rather than models, so their mappings are between theories, translating from the *language* of one sign system to the language of another, instead of just translating the concrete signs in the models. This may sound a bit indirect, but it has important advantages over a model-based approach.

Definition 3.2. 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 $S_1 \to$ predicates and functions 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 defined) $M(c)\colon M(s_1)...M(s_k) \to M(s)$ is a constructor (or function) of S_2,
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, and
4. M is the identity on all sorts and operations for data in S_1.

More generally, a semiotic morphism can map source system constructors and predicates to compound terms defined in the target system (this is illustrated by $M(c)$ is the example just below). $\square$

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

$$M(c(a, f(3,b))) = c'(a', b'+5)+1 \ .$$

A good semiotic morphism should preserve as much of the structure in its source sign system as possible. Certainly it should map sorts to sorts, subsorts to subsorts, data sorts to data sorts, constants to constants, constructors to constructors, etc. But it turns out that in many real world examples, some information is not preserved; so these must all be *partial* maps. Axioms should also be preserved — but again in practice, sometimes not all axioms are preserved. The following preservation properties of semiotic morphisms give systematic ways to measure *representation quality*:

[12] An additional assumption called "reachability" is needed to show the equivalence of these two axioms [19].

[13] Although the root "morph" of "morphism" means "form," this word has recently also become a verb meaning "to change form."

Definition 3.3. Given a semiotic morphism $M: S_1 \to S_2$, then:

1. M is **level preserving** iff the partial ordering on levels is preserved by M, in the sense that if sort s is lower level than sort s' in S_1, then $M(s)$ has lower (or equal) level than $M(s')$ in S_2.
2. M is **priority preserving** iff $c < c'$ in S_1 implies $M(c) < M(c')$ in S_2.
3. M is **axiom preserving** iff for each axiom a of S_1, its translation $M(a)$ to S_2 is a logical consequence of the axioms in S_2.
4. Given also $M': S_1 \to S_2$, then M' is (at least) **as defined as** M, written $M \subseteq M'$, iff for each constructor c of S_1, $M'(c)$ is defined whenever $M(c)$ is.
5. Given also $M': S_1 \to S_2$, then M' **preserves all axioms that** M does, written $M \preceq M'$, iff whenever M preserves an axiom a, so does M'.
6. Given also $M': S_1 \to S_2$, then M' **preserves** (at least) **as much content as** M, written $M \ll M'$, iff M' is as defined as M and M' preserves every selector that M does, where a morphism $M: S_1 \to S_2$ **preserves** a **selector** f_1 of S_1 iff there is a selector f_2 for S_2 such that for every sign x of S_1 where M is defined, then $f_2(M(x)) = f_1(x)$, where
7. a **selector** for a sign system S is a function $f: s \to d$, where s is a sign sort and d a data sort of S, such that there are axioms A' such that adding f and A' to S is consistent and defines a unique value $f(x)$ for each sign x of sort s. For example, each parameter of a constructor has a corresponding selector to extract its value.

□

These ideas are applied somewhat informally in Section 3.6 to justify several basic user interface design decisions for our system. The intuition for 5 is that content is preserved if there is some way to retrieve each data value of the source sign from its image in the target sign system. The definition of selector in condition 6 is technical [26].

It may be that neither M nor M' preserves strictly more than the other; for example, M might preserve more constructors while M' preserves more content. Also, each of these orderings is itself partial, not total. Still other orderings on morphisms than those defined above may be useful for some applications; for example, special measures may be important at certain levels of some signs systems, such as phonological complexity (which is the effort of pronunciation) for spoken language. In general, we should expect that specific "designer orderings," which combine various preservation properties in a specific prioritized way, may be needed to reflect the design tradeoffs of specific applications. The result of all this is that, given sign systems S_1, S_2, we can assume a partial ordering on the collection of semiotic morphisms from S_1 to S_2; this helps to motivate the $\frac{3}{2}$-categories introduced in [26].

Experiments reported in [23] show that preserving high levels is more important than preserving priorities, which in turn is more important than preserving content. They also show a strong tendency to preserve higher levels at the expense of lower levels. This may be surprising, because of emphasis by cognitive psychologists on the "basic level" of lexical concepts (such as "bird"), e.g., [48, 49]. The sentential level of natural language was long considered basic, but research like that of [23] shows that the discourse level is higher in our technical sense, and thus more important. This suggests the general principle that preserving form is more important than preserving content; let us call this **Principle F/C**, where the definitions above allow us to be quite precise in saying that form and content preservation mean preserving constructors and selectors, respectively. Although this principle might seem counterintuitive at first, we will see examples of it in our interface design. The principle really asserts a tradeoff between form and content, where form is more heavily weighted than content, and where the right balance between them can only be determined based on information about how the representation will *actually be used*. It should not be applied blindly. But the principle is worth emphasizing because egregious counterexamples are so often seen, e.g., the internet today is an enormous mass of formless content, most of which is of very low quality. The principle is important for theorem prover interfaces, because the content of proof trees tends towards homogeneity.

Algebraic semiotics has applications other than user interface design, including the theory of metaphor [8, 59], generating good icons, file names, or explanations, and combinging signs from diverse media in appropriate ways. See [26] for more detail, and see the "UC San Diego Semiotic Zoo" for examples of bad design that arise from failures of semiotic morphisms to preserve some important structure (at `www.cs.ucsd.edu/groups/zoo`).

3.2. ProofWebs and ProofDags

The signs that users actually see are called **ProofWebs**. For example, Figure 4 gives screendumps of a typical homepage and proof page. ProofWebs consist of colored shapes, navigation buttons, etc., and should be considered an experiential category, rather than a formal data structure. The tatami conventions (Section 3.4) are our design guidelines for these complex signs.

However, the structure of ProofWebs can be captured at various levels of abstraction by formal abstract data types (ADTs). The abstract structure that is closest to actual ProofWebs is captured by the XML code from which they are generated by Kumo. An algebraic specification of some aspects of this ADT is given in Section 3.5 using OBJ3; hopefully it is not too confusing to also use the name "ProofWeb" for this more abstract structure. We use this ADT in showing that some status window designs are better than others. Figure 5 shows the abstract structure of a small ProofWeb expressed in the language of this algebraic specification, and Figure 6 shows the abstract structure of a small status window.

At a still more abstract level, **ProofDag**s abstract out the information that is most important for distributed proofs ("dag" stands for "directed acyclic graph," a generalization of tree that allows some nodes to be shared). This data structure is used in [22] to give a precise description of the data for which logical consistency is maintained by the tatami protocol, and it is used in the proof of its validity on our website. A complication we do not consider here is its "2-dimensional" structure, used to keep track of alternative proofs for the same goal [22].

3.3. Narratology and Cinema

In order to help ordinary software engineers, we should make specifications and proofs as understandable, and even interesting, as possible. Typical "modern" mathematical proofs hide the often considerable conflicts that were involved in their construction; finding a non-trivial proof usually requires exploring many

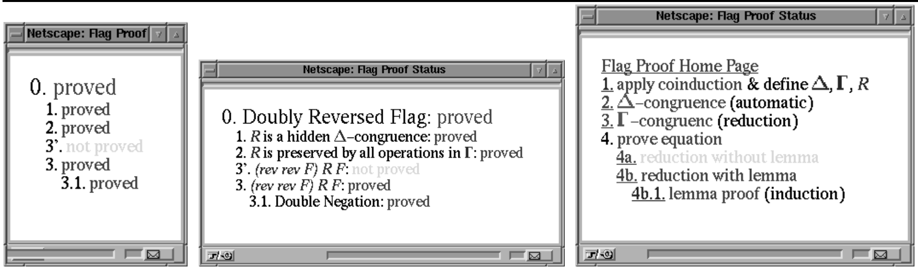

Fig. 3. Screendumps of Three Status Window Designs

misconceptions and errors, some of which may be very subtle (though many will be trivial). Published proofs bury all this, exhibiting only the tricks that finally conquered the obstacles, without showing why those tricks were needed.

We suggest an approach (which might be called "postmodern") to making proofs more understandable and interesting by recording in ProofWebs the conflicts that motivate difficult proof steps; as Aristotle said, "*Drama is conflict.*" Structuring proofs this way should make them easier to understand, although of course it would be counterproductive to record every small error that was made in the proving process. The work of Joseph Campbell [2] and Christopher Vogler [60] on the role of characters in stories, especially heroes, is also relevant; the importance of having the hero tested by obstacles is emphasized in these works, and we have used in to structure our flag proof website (item 2 in Section 2). The role given to archetypal characters is also suggestive, but we have yet to explore its implications for theorem prover interface design.

Another resource that we have used in the theory of stories. William Labov [39] shows that oral narratives have a precise structure, involving a sequence of "narrative clauses" describing events whose default ordering 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 (see the discussion in [18]); there are also an optional opening "orientation" section and an optional closing section; the former gives necessary background for the story, such as time and place, while the latter may give a summary and/or "moral" for the story as a whole. These ideas have had a major influence in designing the ProofWebs generated by Kumo (see Section 3.6). Further study of how evaluative material connects to narrative material may give further ideas for structuring proof motivation. For example, Syd Field's screen writing books (e.g., [9]) give a precise but naive dramatic structure for Hollywood plots: they should have three acts, for setup, conflict, and resolution, with "plot points" that move action from one act to the next. The study of embedded stories, flashbacks, and so on might also be useful in making proofs easier to understand.

3.4. The Tatami Conventions

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 special application involved (e.g., see [47, 53]).

We have developed the following **tatami conventions** as style guidelines for the proof websites generated by Kumo:

1. The main **proof pages** describe proof steps in relatively small groups (about seven non-automatic rules per page works well). Proof pages appear in a fixed master window; see the right side of Figure 4.
2. Proof pages can be browsed in a "narrative" order, designed by the prover to be helpful and interesting to the reader; if possible, they should tell a story about how obstacles were overcome (or still remain).
3. A user-supplied informal explanation page is linked to each proof page, discussing the proof concepts, strategies, obstacles, etc. for that page; these can have graphics, applets, and of course text; they appear in their own 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 proof pages; see the left side of Figure 4[14]. A dedicated persistent popup window for lemmas has the same structure as the master window.
5. Major proof parts can have their own "closing" webpage to sum up results and lessons; they appear in the same window as their proof pages.
6. Formal proof steps are automatically hotlinked to pre-existing tutorial background pages; e.g., each application of induction is linked to a webpage that explains the kind of induction used. The tutorial pages have a dedicated persistent popup window.
7. A formal proof script is generated for each proof page; proof readers can view them on a dedicated persistent popup, and request execution on a proof server, with result displayed in the same window as the script.
8. Each kind of webpage has its own distinctive background and frame; the frame provides navigation buttons 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 (see Figure 3).

These conventions have the effect of integrating proofs with the information that is needed to understand and debug them. Section 3.6 will justify many design decisions, using ideas mainly from narratology and semiotic morphisms.

Let's look at proof pages and status windows in a little more detail. The navigation buttons for proof pages are $\boxed{\text{UP}}$, $\boxed{\text{DOWN}}$, $\boxed{\text{LEFT}}$, and $\boxed{\text{RIGHT}}$, plus $\boxed{\text{PREV}}$ and $\boxed{\text{NEXT}}$; the first four support traversal of the acyclic proof graph structure, while the latter two follow the "narrative order" that has been defined by the prover; see the right side of Figure 4. Proof structure is also displayed in a **status window** popup, which is activated when the uses pushes the $\boxed{\text{STATUS}}$ button. Status values are updated by propagating changes up the proof tree; for completely formal proofs, the status can be `proved`, `unknown`, or `untrue`. The status window also supports proof navigation by clicking on the desired description; Figure 3 shows screendumps of three different status window designs.

[14] This figure actually shows an obsolete version, which will be used in Section 3.6 to illustrate the use of semiotic morphisms to critique inference design.

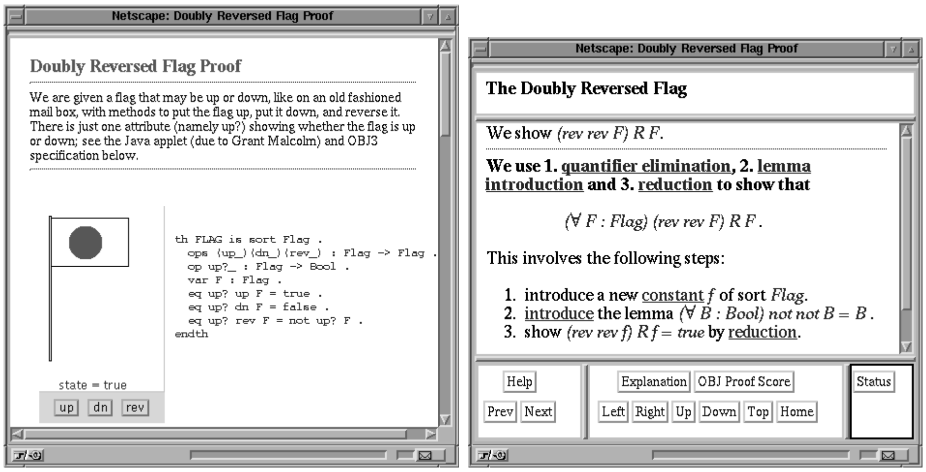

Fig. 4. Screendumps of a Typical Homepage and Proof Page

3.5. ProofWeb and Status Window ADTs

The first three OBJ3 modules below give a (partial) algebraic specification for the ProofWeb sign system. For concision we use `T`, `?` and `F` for proof status values, instead of `proved`, `unknown`, or `untrue`; if we were also allowing informal validations, these would instead be fuzzy truth values.

OBJ3 [25, 32] modules that are to be interpreted loosely begin with the keyword `th` and close with the keyword `endth`. Between these two keywords come declarations for sorts and operations, plus variables and equations. Any number of sorts can be declared following `sorts` (or equivalently, `sort`), and operations are declared with their arity between the `:` and the `->`, and with their value sort following the `->`. The keyword `pr` indicates that the named module expression is to be imported. The underbars in operator declarations indicate where arguments go, for mixfix syntax. The keyword `dfn` is shorthand for importing a module expression of the form `Mod *(Ppl to Name)`, where `Mod` is the module at the end of the definition, `Name` is the name given at the beginning, and `PPl` is the principal sort of `Mod`; this indicates that the principal sort of `Mod` is to be renamed.

```
th TATAMI-BASE is sorts Pf TStat .
  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 T ? F : -> TStat .
endth
th TATAMI-PF is pr TATAMI-BASE .
  sort HPg .  dfn Goal is QID .  dfn Rule is QID .
  op HPg_[___] : Name Goal Pf TStat -> HPg .
  op PfOf_By_[___]_ : Goal LRule ExPg LPf Obj TStat -> Pf .
  op ALT : LPf -> Pf .
  op subg : Pf -> Rule .
endth
th TATAMI is pr TATAMI-PF .
  sort Tatami .
  dfn LHPg is LIST[HPg].
```

```
  op Tat  : LHPg -> Tatami .
endth
```

The top level signs of this system have sort `Tatami` with primary (and only) constructor `Tat`, which takes one argument of sort `LHPg` (for lists of `HPg`s), which in turn has as its primary constructor the concatenation operation for proof homepages (there is also a secondary constructor for `LHPg`, namely the empty list, denoted `[]`). `HPg` has primary (and only) constructor `HPg`, with arguments of sorts `Name`, `Goal`, `Pf` and `TStat`; technically these are fourth level sorts, but since `Tatami` and `Hpg` each have just one unary constructor, to avoid level proliferation, let's agree to call all these sorts second level. The most interesting one is `Pf`, for proofs, with primary constructor `PfOfBy`; this has `LPf` (for lists of proofs) as an argument sort; `Pf` also has a secondary constructor `ALT` with `LPf` as its only argument sort; this allows multiple proof attempts for a single goal. Therefore `Pf` is a recursive sign sort in two different ways; it is the sort of our proof pages. The third level sorts are `LRule`, `ExPg` and `Obj`, for rule list, explanation page, and OBJ code, respectively. Finally, `Rule` is a fourth level sort, with constructor `subg` which allows introducing new results (usually called "lemmas"). Since these still must be proved, `subg` has argument sort `Pf`, giving another recursion.

Next we give the structures of two proofs that are on the web; Figure 5 shows them as trees, `flag` and `sum`, which are respectively the left and right subtrees of the main tree. The keyword pair `obj`, `endo` indicates that initial semantics is being used, i.e., that the standard model is desired. The operations with `Nat` as argument just provide a collection of named objects of their target sort. The `let` construct abbreviates the combination of a declaration for a new constant, and an equation defining that constant to be some term.

```
obj TESTAMI is pr TATAMI .
  op ex : Nat -> ExPg .   op pf  : Nat -> Pf .
  op g  : Nat -> Goal .   op obj : Nat -> Obj .
  let flag = HPg('flag)[g(1) pf(1) T] .
  let sum  = HPg('sum) [g(2) pf(2) T] .
  let t    = Tat(flag sum).
  eq pf(1)  = PfOf g(1)  By('coind) [ex(1) (pf(11) pf(12)
                                ALT(pf(13) pf(131))) obj(1)] T .
  eq pf(13) = PfOf g(13) By('red) [ex(13) (nil) obj(13)] ? .
  eq pf(131)= PfOf g(13) By(subg(pf(14)) 'red) [ex(131) (nil) obj(131)] T .
  eq pf(2)  = PfOf g(2)  By('ind) [ex(2) (pf(21) pf(22)) obj(2)]  T .
  eq pf(21) = PfOf g(21) By('qelim 'red) [ex(21) (nil) obj(21)] T .
  eq pf(22) = PfOf g(22) By('qelim 'ielim 'red) [ex(22) (nil) obj(22)] T .
endo
red sum .  red flag .  red t .
```

Running the above `red` (for reduce) commands gives exactly (terms for) the trees shown in Figure 5, where the large dot nodes indicate the list concatenation operation. Note that the `flag` structure has an `ALT` constructor with two proofs for the third step of the coinduction; the first uses just reduction and fails (giving truth status `?`), while the second succeeds by making use of a lemma.

We now (partially) specify the status window structure used in our current Kumo prototype, and again give the flag proof as an example. Note the `ALT` constructor in the first module below.

```
th STATWIN is pr TATAMI-BASE .
  sorts SWin SItem .
  dfn LSItem is LIST[SItem].  dfn LLSItem is LIST[LSItem].
  op Swin_[_]_  : Name   SItem TStat -> SWin .
```

```
  op SItem_[_]_ : LRule LSItem TStat -> SItem .
  op ALT :  LSItem -> SItem .
  op subg : SItem  -> Rule .
endth
obj TESTSWIN is pr STATWIN .
  op w : Nat -> SItem .
  let wflag = Swin('flag)    [w(1)] T .
  eq w(1)   = SItem('coind) [w(11) w(12) w(13)] T .
  eq w(12)  = SItem('red)   [nil]  T .
  eq w(13)  = ALT((SItem('red) [nil] ?) (SItem(subg(w(14)) 'red) [nil] T)).
endo
red wflag .
```

Finally we define a "view" from single ProofWebs (i.e., the `TATAMI-PF` spec, not the `TATAMI` spec) to status windows, describing a *partial* semiotic morphism from `TATAMI` to `STATWIN`, as is typical, and in fact for this application is necessary, because the status window only displays information for one proof at a time. It is very convenient that OBJ's view features corresponds exactly to semiotic morphisms (see [25, 32] for more on views in OBJ).

```
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 : TStat .
  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
```

3.6. Justifying Design Decisions

This section applies techniques discussed earlier in this paper to design decisions for the proof websites produced by Kumo[15]. We first comment on the tatami conventions, using the same enumeration as in Section 3.4; many of these justifications draw on narratology.

1. Limiting the number of non-automatic proof steps on proof pages is consistent with limitations of human cognitive capacity, as made famous by George Miller in his "magic number seven plus or minus two" paper [42]. Automatic proof steps (like conjunction elimination) place a lower cognitive load on the reader than those, like lemma introduction, where the prover had to supply additional information.
2. The idea of a giving a "narrative" order to proof pages comes from the theory of stories [39]; the idea of including obstacles comes from Campbell [2] and others (going back to Aristotle).
3. Attaching user-supplied informal explanation pages to proof pages was suggested by the close connection between narrative clauses and evaluative material in stories [39]; the evaluative material provides the motivation that is needed for the important steps in the proof, by relating them to values shared among provers. Placing these in a separate window parallels the syntactic structure used in stories.

[15] Some of the design decisions were guided in advance of construction by theory, but in other cases, the theory was mobilized after the fact to better understand decisions or to suggest further improvements.

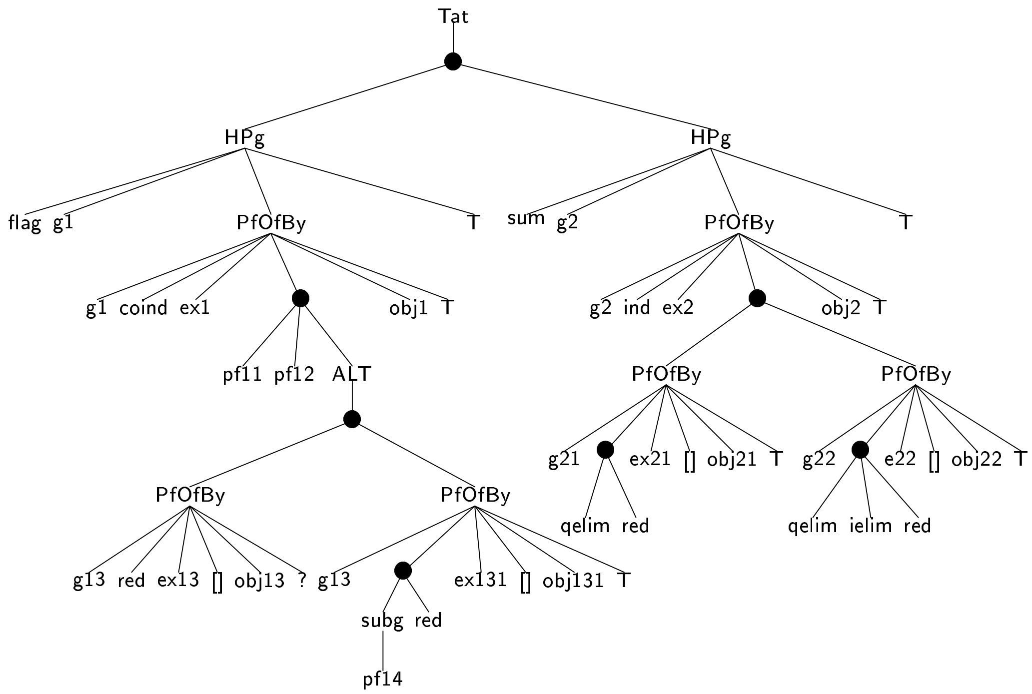

Fig. 5. Tree Structures of Two Simple ProofWebs

4. Using homepages for major proof parts is motivated by the opening "orientation" sections in Labov's theory of story structure [39]; they appear in the same window as proof pages, since 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 [39], 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, e.g., [50].
7. Separating formal proof scripts from the proof pages that generated them allows hiding the most routine details of proofs, just as human proofs often omit details in order to highlight the main ideas [40]; however, proof readers can still view them, and even execute them on a proof server. This is justified by Principle F/C (see below for details).
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 leaf nodes of the current proof tree, and hence form a list in a natural way. Omitting all other content is now an instance of Principle F/C; similar considerations apply to the status window design.

The remaining design decisions are justified using algebraic semiotics. The basis for these arguments is that any display to users of information in the

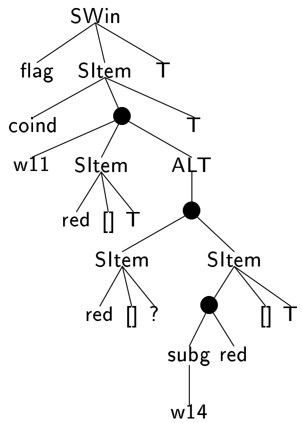

Fig. 6. Tree Structure of a Status Window

system can be seen as a morphism from a sign system for ProofWebs into a sign system for webpage functionality. For the sake of brevity and clarity, we will be somewhat informal about both of these sign systems in this discussion, noting that the `TATAMI` sign system given in Section 3.5 leaves out some structure that is important for the present discussion, especially the clustering of proof steps into proof pages.

Windows: The main content bearing sorts of ProofWebs are for proof steps, informal explanations, tutorials, and executable proof scripts. These four sorts are the main arguments of the highest priority constructor `PfOfBy` in the `TATAMI-PF` theory of Section 3.5, and their preservation is important for the quality of a representation. Proof pages are the main constituents of a ProofWeb, and hence theirs is the master window. Explanation pages have their own window, which is a persistent popup, triggered by a button on the master window, and dismissable by the proof reader with a button in its own frame; the same holds for tutorial and proof script pages. There is also a persistent dismissable popup for lemmas, having the same structure as the master window. (As a bonus, the browser's BACK button retrieves information previously seen in each kind of window.)

Backgrounds: Each kind of window has its own distinctive background: proof pages have a tatami mat background, explanation pages have a pink marble background, tutorial pages have a yellow marble background, and proof script pages have a blue raindrop background. This again reflects the importance and distinctness of the four argument sorts of the highest priority constructor, 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 ProofWeb; this name appears as an argument to the highest level constructor `HPg`; the title frame also contains the name of the current node. 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.

As the project progressed, different versions of the status window were developed; three of these are shown in Figure 3. We now show that these obsolete versions do *not* admit semiotic morphisms from the ProofWeb sign system of Section 3.5. The left display gives very little information about the proof, and represents `ALT` using a prime for the failed subproof; this notation does not generalize to multiple attempts where some fail and others succeed. The middle display gives more information, e.g., goals, but represents `ALT` the same way as the left display, and fails to indicate the initial (coinduction) proof step; giving goals is not a good idea, because these can be very complex; this is an instance of Principle F/C. Both displays have redundant proof status information, e.g., "proved" and the color green[16]; neither indicates the proof homepage. The right display gives generic proof step names instead of goals and lists the proof rules[17]; but it also represents the initial step the same way as its three subgoals, and thus fails to preserve the `PfOfBy` constructor. Thus none of these three displays can be an instance of a semiotic morphism from the `TATAMI` spec, because in each case some key constructor is not preserved. A similar argument can be given for the proof page design shown in the right side of Figure 4: first giving an enumeration of proof rules and then using it to enumerate proof steps does not extend well to proofs where rules are used multiple times; this corresponds to non-preservation of the `PfOfBy` constructor, since the design tries to map a tree into a pair of lists plus cross-referencing.

Here is a somewhat different application of semiotic morphisms: Section 3.3 noted that Field [9] suggests that movies should have a three part structure, for setup, conflict, and resolution. We have experimented with proofs having this structure, and found that they are easier to understand; in fact, the flag ProofWeb has this structure, where conflict arises from the failure of a reduction, and is resolved when the appropriate lemma is provided (see Figure 3). We can consider that this structure is imparted by a semiotic morphism from a three part "Syd Field" sign system to the ProofWeb.

4. Experience and Conclusions

Some simple experiments in cooperative distributed proving were done on an earlier prototype implementation. Although simple, these experiments still involved nontrivial decisions, such as choosing a candidate relation for coinduction, introducing lemmas, and using case analysis. There was little trial and error with the proofs in these experiments, because the subjects were already familiar with the examples. But interesting observations were still made concerning the coordination of the cooperative work involved (see below).

We also did two experiments in distributed cooperative verification using

---

[16] Figure 3 has black for the original green representing "`T`", and gray for original yellow representing "`?`".

[17] In practice, a filter is needed if there are many rules. This is another example of Principle F/C: it is important to preserve proof structure as support for proof overview and navigation; therefore if long lists of rules interfere with the clarity of structure, then some rules should be eliminated from the display.

email to exchange specs and proofs [17]. The more complex of these involved a total of 36 messages; participants were in four different countries, each in a different time zone. We were surprised to discover that in both experiments, a majority of the changes suggested by participants involved a change to the specification, not just to the proof, and many involved reformulating the goal. Although this is not a new result, it does run contrary to current culture in the formal methods community, which has generally (implicitly) assumed a fixed specification and a fixed goal, with all the effort devoted to getting the proof right; in fact, formal methods systems tend to make the even stronger assumption of step-wise software development, i.e., the discredited waterfall model. Combining this with our more extensive single user experiments suggests the following conclusions:

- Because of frequent changes to specs and goals, the system should support flexible user interactions, including co-evolution of proofs, specs and goals, starting in the middle of a proof, and leaving parts unfinished.
- Relations among specifications, including enrichment, equivalence, and refinement, are important and should be tracked in the database.
- Communication is vital for the early stages (determining the specification, goal and proof strategy); a real time chatroom does not seem promising, because users can have very different schedules, but a MUD (Multi User Dungeon) that retains comments should be tried.
- Behavioral proof methods like coinduction help avoid far more complex arguments of a more conventional kind.
- Distributed cooperative work should be supported, with rapid communication and multimedia interfaces, allowing users to achieve transparency with respect to locations of documents and (to some extent) of other users.
- A good interface to the specification database is important, because users can easily get lost when there are many specifications having complex relationships, such as enrichment. The module graph concept of [31] provides one way to organize this complex information, but navigation issues have as yet been little explored.
- It can be hard to know what other users have done, because demons automatically update the databases; some kind of "What's New" feature should be developed and evaluated.
- It is important to support informal and semi-formal validation, because fully formal proofs are rarely worth the trouble in practice.

Although these points are unlikely to surprise experienced software engineers, they are all contrary to current practices and ideologies in many parts of the formal methods community; especially the last point is violently opposed by some, and there is also a strong bias against collaboration in formal verification, perhaps because mathematical training emphasizes individual achievement, competition, and complete formal rigor.

The above observations played a key role in planning the next generation system now under development.

This paper has tried to present evidence that some semantic issues in user interface design can be dealt with in a direct way, that avoids the tedious and expensive methods of experimental psychology, and that also avoids the use of *ad hoc* assumptions or reliance on experience with prior systems that may be only remotely related to the current concerns. In any case, we have found techniques from ethnography, discourse analysis, cognitive science, and algebraic semiotics very helpful both in the initial design and subsequent improvement of our style guidelines for proof websites. We hope that others may also benefit, as well as contribute further to the development of these still rather nascent ideas.

Acknowledgements

I thank the members of the "links" group at UCSD for their support and hard work during the laborious development of the material described in this paper. Kai Lin has been responsible for most of the implementation work, including all of Kumo, and many of the proofs that use it; Grigore Roşu has also done some proofs, has designed the tatami protocol, and has helped with many theoretical developments; Dr. Akira Mori helped get the project started with hand-made versions of websites that were later generated by Kumo, and he helped with the figures for this paper; with Akiyoshi Sato, Dr. Mori also wrote some daemons for the protocol implementation; Prof. Eric Livingston helped with discussions on social issues. I thank them all, as well as Prof. Kokichi Futatsugi for encouragement and support through the CafeOBJ project in Japan.

References

[1] Robert Bailey. *Human Performance Engineering.* Prentice-Hall, 1996.

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

[3] Manuel Clavel, Steven Eker, Patrick Lincoln, and José Meseguer. Principles of Maude. In José Meseguer, editor, *Proceedings, First International Workshop on Rewriting Logic and its Applications.* Elsevier Science, 1996. Volume 4, *Electronic Notes in Theoretical Computer Science.*

[4] CoFI. CASL summary, 1998. `http://www.brics.dk/Projects/CoFi/`.

[5] 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.

[6] Răzvan Diaconescu and Kokichi Futatsugi. Logical semantics for CafeOBJ. Technical Report IS–RR–96–0024S, Japan Advanced Institute for Science and Technology, 1996.

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

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

[10] Kokichi Futatsugi, Joseph Goguen, Jean-Pierre Jouannaud, and José Meseguer. Principles of OBJ2. In Brian Reid, editor, *Proceedings, Twelfth ACM Symposium on Principles of Programming Languages*, pages 52–66. Association for Computing Machinery, 1985.

[11] Kokichi Futatsugi and Ataru Nakagawa. An overview of Cafe specification environment. In *Proceedings, ICFEM'97.* University of Hiroshima, 1997.

[12] Harold Garfinkel. *Studies in Ethnomethodology.* Prentice-Hall, 1967.

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

[14] Joseph Goguen. Principles of parameterized programming. In Ted Biggerstaff and Alan Perlis, editors, *Software Reusability, Volume I: Concepts and Models*, pages 159–225. Addison Wesley, 1989.

[15] Joseph Goguen. On notation (a sketch of the paper). In Boris Magnusson, Bertrand Meyer, and Jean-Francois Perrot, editors, *TOOLS 10: Technology of Object-Oriented Languages and Systems*, pages 5–10. Prentice-Hall, 1993. The extended version of this paper may be obtained from `http://www.cs.ucsd.edu/users/goguen/ps/notn.ps.gz`.

[16] Joseph Goguen. Requirements engineering as the reconciliation of social and technical issues. In Marina Jirotka and Joseph Goguen, editors, *Requirements Engineering: Social and Technical Issues*, pages 165–200. Academic, 1994.

[17] Joseph Goguen. An empirical study of distributed cooperative verification, 1997. Unpublished draft manuscript, Dept. Computer Science & Engineering, UCSD.

[18] 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.

[19] 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.

[20] Joseph Goguen. First order logic. In *Theorem Proving and Algebra*. MIT, to appear. This chapter can be obtained at the URL `http://www.cs.ucsd.edu/users/goguen/ps/tp/fol.ps.gz`.

[21] Joseph Goguen and Kokichi Futatsugi. Semiotic redesign of a specification language, in preparation.

[22] 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.

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

[24] Joseph Goguen and Charlotte Linde. Techniques for requirements elicitation. In Stephen Fickas and Anthony Finkelstein, editors, *Requirements Engineering '93*, pages 152–164. IEEE, 1993. Reprinted in *Software Requirements Engineering (Second Edition)*, ed. Richard Thayer and Merlin Dorfman, IEEE Computer Society, 1996.

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

[26] 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.

[27] Joseph Goguen and Grant Malcolm. More higher order programming in OBJ3. In Joseph Goguen and Grant Malcolm, editors, *Software Engineering with OBJ: Algebraic Specification in Action*. Kluwer, to appear.

[28] Joseph Goguen and José Meseguer. Order-sorted algebra I: Equational deduction for multiple inheritance, overloading, exceptions and partial operations. *Theoretical Computer Science*, 105(2):217–273, 1992. Drafts exist from as early as 1985.

[29] Joseph Goguen, James Thatcher, and Eric Wagner. An initial algebra approach to the specification, correctness and implementation of abstract data types. In Raymond Yeh, editor, *Current Trends in Programming Methodology, IV*, pages 80–149. Prentice-Hall, 1978.

[30] Joseph Goguen, James Thatcher, Eric Wagner, and Jesse Wright. Initial algebra semantics and continuous algebras. *Journal of the Association for Computing Machinery*, 24(1):68–95, January 1977. An early version is "Initial Algebra Semantics", by Joseph Goguen and James Thatcher, IBM T.J. Watson Research Center, Report RC 4865, May 1974.

[31] Joseph Goguen and William Tracz. An implementation-oriented semantics for module composition. In Gary Leavens and Murali Sitaraman, editors, *Foundations of Component-based Systems*. Cambridge, 1999. To appear.

[32] 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, to appear. Also Technical Report SRI-CSL-88-9, August 1988, SRI International.

[33] John Guttag, James Horning, and Jeanette Wing. Larch in five easy pieces. Technical Report 5, Digital Equipment Corporation, Systems Research Center, July 1985.

[34] Lutz Hamel. *Behavioural Verification and Implementation of an Optimizing Compiler for OBJ3.* PhD thesis, Oxford University Computing Lab, 1996.

[35] Masako K. Hiraga. Diagrams and metaphors: Iconic aspects in language. *Journal of Pragmatics*, 22:5–21, 1994.

[36] Brian Jepson and David Hughes. *Official Guide to Mini SQL 2.0.* Free Press, 1995.

[37] William Labov. *Language in the Inner City.* University of Pennsylvania, 1972.

[38] William Labov. *Sociolinguistic Patterns.* University of Pennsylvania, 1972.

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

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

[41] 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.

[42] 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.

[43] Peter Padawitz. Towards the one-tiered design of data types and transition systems. In *Proceedings, WADT'97*, pages 365–380. Springer, 1998. Lecture Notes in Computer Science, Volume 1376.

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

[45] Charles Saunders Peirce. *Collected Papers.* Harvard, 1965. In 6 volumes; see especially Volume 2: Elements of Logic.

[46] Marian Petre and Blaine Price. Why computer interfaces are not like paintings: the user as a deliberate reader. In *Proceedings, East-West HCI'92, Vol. I*, pages 217–224. Int. Centre for Scientific and Technical Information, Moscow, 1992.

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

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

[49] Eleanor Rosch. Cognitive reference points. *Cognitive Psychology*, 7, 1975.

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

[51] Harvey Sacks. *Lectures on Conversation.* Blackwell, 1992. Edited by Gail Jefferson.

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

[53] Ben Shneiderman. *Designing the User Interface.* Addison Wesley, 1997.

[54] Ian Sommerville. *Software Engineering.* Addison-Wesley, 1995. Fifth edition.

[55] Y.V. Srinivas and Richard Jüllig. SpecWare language manual, version 2.0. Technical report, Kestrel, 1996.

[56] Grigore Roşu and Joseph Goguen. Hidden congruent deduction. In Ricardo Caferra and Gernot Salzer, editors, *Proceedings, 1998 Workshop on First Order Theorem Proving*, pages 213–223. Technische Universität Wien, 1998. (Schloss Wilhelminenberg, Vienna, November 23-25, 1998). Full version to appear in *Lecture Notes in Artificial Intelligence*, Springer, 1999.

[57] William Tracz. LILEANNA: a parameterized programming language. In *Proceedings, Second International Workshop on Software Reuse*, pages 66–78, March 1993. Lucca, Italy.

[58] William Tracz. *Formal Specification of Parameterized Programs in* LILLEANNA. PhD thesis, Stanford University, 1997.

[59] Mark Turner. *The Literary Mind.* Oxford, 1997.

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

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

A. A Sample Duck Script

A script for proving the correctness of the array-with-pointer implementation of stack is given below in the Duck language. This text is divided into two parts, the first for proofs and the second for display. In addition to the main goal, there are two lemmas. Each proof begins with the keyword "`proof:`" and ends with "`[]`", indicating that Kumo should try to finish the proof using its built in tactics[18], e.g., reduction and elimination for universal quantifiers, conjunction, etc. Comment lines begin with "`***`". A name is optionally given after "`proof:`". Following that comes a "`goal:`", and then the proof steps, after "`by:`". Here the lemmas are proved by induction, while the main goal is proved by coinduction; however, we will not go into the details.

In the display part, the file named ater "`markup:`" defines the XML macros used, and the phrase given after "`title:>`" is displayed at the top of each generated proof page, while that after "`subtitle:>`" gives a headline for a proof page. The address after "`putdir:>`" gives the directory (in the document area of the local web server) where the generated HTML files will be placed. The informal explanation pages attached to proof pages are fetched from default locations in this example, but explicit URLs can also be given. Note the use of names for proof parts, and of a default name convention when explicit names are not used.

```
*** ***********************************************************
*** proof score for stack refinement
*** file: /net/cat/disk1/goguen/duck/stack.duck
*** ***********************************************************
spec: /kumo/stack/stack
*** ***********************************************************
proof: <<StackRepLemma>>
  goal: (forall S : Stack)(exists I : Nat)(exists A : Arr) S = I || A .
  by: induction on S with {empty, push}  []
*** ***********************************************************
proof: <<PutAndBar>>
  goal: (forall I I1 I2 : Nat)(forall A : Arr)
          (not (I2 < I1) implies  I1 || put(I,I2,A) = I1 || A).
  by: induction on I1 with scheme {0, s}  []
*** **********************************************************
proof: <<StackThm>>
  goal: pop empty = empty and
      ((forall N I : Nat forall A : Arr) top push(N, I || A) = N) and
      ((forall N I : Nat forall A : Arr) pop push(N, I || A) = (I || A)).
relation:
   op _R_ : Stack Stack -> Bool .
   var I1 I2 I : Nat .
   var A1 A2 A : Arr .
   eq (s I1 || A1) R (I2 || A2) =
         I2 == s I1 and A1[I1] == A2[I1] and (I1 || A1) R (I1 || A2) .
   eq (0 || A1) R (I || A2) = I == 0 .
   eq (I || A)  R (I || A)  = true .
[]
cobasis: {top, pop}  *** system could compute this as default
by: coinduction
 <<StackThm.1>>
    instantiate S1 S2 with StackRepLemma;
    uq-elim;
```

[18] This design decision was inspired by Larch [33]; the underlying functionality is similar to that of the "`grind`" command of PVS.

```
    <<:= DeltaCase>> case-anal on ii(s1) with NatRepLemma;
      <<*.2>>
        imp-elim;
        skolemize with NatRepLemma
    [StackThm.1]
 <<*.2>>
    instantiate S1 S2 with StackRepLemma;
    uq-elim;
      <<:= GammaCase>> case-anal on ii(s1) with NatRepresent
        <<*.2>>
          imp-elim;
          skolemize with NatRepLemma;
          lemma PutAndBar
[]
*** ***********************************************************
display:
  title: "Array Representation of Stack"
  putdir: /net/cs/htdocs/groups/tatami/demos/stackrep
  markup: ~/myMarkup
<<StackRepLemma>>
  subtitle: "We show that any stack can be represented by an array"  []
*** **********************************************************
display:
  title: "Key Lemma for Stack Implementation "
  putdir: /net/cs/htdocs/groups/tatami/demos/putandbar
<<PutAndBar>>
  subtitle: "We prove the key lemma: "  []
*** **********************************************************
display:
  title: "Array-with-Pointer Implementation of Stack"
  putdir: /net/cs/htdocs/groups/tatami/demos/stack
<<StackThm>>
  subtitle: "We show that the array-with-pointer implementation of "+
             "stack is correct."
<<*.1>>
  subtitle: "We show <math>R</math> is a hidden <Delta/>-congruence."
  <<DeltaCase.1>>
    subtitle: "Case1: <math>ii(s1) = 0</math>."  [;]
  <<DeltaCase.2>>
    subtitle: "Case2: <math>ii(s1) = s(i)</math>."
<<StackThm.2>>
  subtitle: "We show <math>R</math> preserved by all operations in <Gamma/>."
  <<GammaCase.1>>
    subtitle: "Case1: <math>ii(s1) = 0</math>."  [;]
  <<GammaCase.2>>
    subtitle: "Case2: <math>ii(s1) = s(i)</math>."
<<StackThm.3>>
  subtitle: "Finally we prove the <math>Stack</math> equations."
[]
```

The output website that this produced can be viewed at `www.cs.ucsd.edu/groups/tatami`.

B. Words of the Masters

The desire to make mathematics easier with better notation has a long history, as shown by two quotations below. The first[19] is from Gottfried Wilhelm Leibniz:

[19] Unfortunately, I have lost track of the source for this quote.

In signs, one sees an advantage for discovery that is greatest when they express the exact nature of a thing briefly and, as it were, picture it; then, indeed, the labor of thought is wonderfully diminished.

One example is the difference between doing proofs in plane geometry with diagrams and doing them with formal logic and axioms, since the diagrams "picture" the entities and relationships involved both briefly and exactly; another example is the difference between proofs done in words and proofs done using algebraic notation.

The second quote is an extract from a letter from Renee Descartes to Girard Desargues, dated 19 June 1639, quoted in *The Geometrical Work of Girard Desargues*, by J.V. Field and J.J. Gray (Springer, 1986, pages 176–177):

If [it] is your intention ... to write for people who are interested but not learned, and make this subject, which until now has been understood by very few people, but which is nevertheless very useful ..., accessible to the common people and easily understood by anyone who studies it from your book, then you must steel yourself to ... explain everything so fully, so clearly and so distinctly that these gentlemen, who cannot study a book without yawning and cannot exert their imagination to understand a proposition of Geometry, nor turn the page to look at the letters on a figure, will not find anything in your discourse which seems to them to be less easy of understanding than the description of an enchanted palace in a novel.

Could it be that Descartes was thinking along lines similar to those of Section 3.3?