View metadata, citation and similar papers at core.ac.uk

brought to you by CORE

provided by Elsevier - Publisher Connector

Available online at www.sciencedirect.com

SCIENCE DIRECT®

Electronic Notes in Theoretical Computer Science

ELSEVIER Electronic Notes in Theoretical Computer Science 103 (2004) 81–103

www.elsevier.com/locate/entcs

Formalizing the User's Context to Support User Interfaces for Integrated Mathematical Environments

Joseph R. Kiniry[1]

Computing Science Department
University of Nijmegen
Toernooiveld 1
6525 ED Nijmegen
The Netherlands

Abstract

This paper describes the several user-interface features for interactive theorem provers. Many of these features mimic functionality that already exists, and have great utility, in modern interactive development environments (IDEs). A formal kind theoretic model of a user's context is also presented. This model is used to formally describe the structure, behavior, and customization of the features. The functionality presented include browsers for basic mathematical constructs (declarations, theories, types, proofs, etc.), quick access to constructs definitions and uses (a short-cut sidebar, menus, or implicit hyperlinks), built-in contextual help, context- and type-aware completion and visual representation (expanding and collapsing structured elements of specifications, proof terms, and sequents), the graphical representation of language elements, and a user-extensible, type-aware pretty-printer. Research opportunities in interface design based upon the formal model are also identified and discussed. These features have been added to the PVS theorem prover as a proof-of-concept and will be available in its next major release.

Keywords: Interactive theorem prover, interactive development environment, PVS theorem prover.

1 Introduction

The user interfaces (UIs) of modern, popular theorem provers provide all the core functionality necessary to write specifications and perform proofs. Un-

[1] Email:kiniry@cs.kun.nl

1571-0661 © 2004 Elsevier B.V. Open access under CC BY-NC-ND license.
doi:10.1016/j.entcs.2004.09.015

surprisingly, most of these features focus on *mathematics*, rather than *mathematicians.*

The complexity of the theories and specifications attempted in provers has risen dramatically in recent years. For example, the formal semantics of programming languages like Java has been specified in several provers [2]. The proof scripts in one of these efforts [9] (to which the author contributed) are tens of thousands of lines long, and hundreds of strategies have been written to help prove thousands of theorems.

In contrast, only the most *advanced* theorem prover UIs provide features approximating those of *rudimentary* programming environments of over a decade ago, e.g., features like symbol completion, syntax highlighting, and basic documentation lookup.

As a result, theorem prover users, especially those that are also programmers, are now demanding many of the features of modern integrated development environments (IDEs), particularly those features that help manage complex software systems.

The generic conceptual and technical facilities for handling complexity in advanced programming environments, like those available for the C++ and Java programming languages, are also applicable to the UIs of modern theorem provers. To that end, this paper discusses some initial work in improving the user interface of PVS, a higher-order theorem prover available from SRI, through the addition of such facilities.

To formally and generically represent the UI features discussed herein, and as a primary contribution of this work, the definition and elucidation of a new concept called the *semantic context* is proposed. To clearly differentiate technologies, we call an interactive theorem prover (ITP) with a first-class notion of a semantic context an *interactive mathematical environment* (IME) in the following.

1.1 Semantic Contexts

The general notion behind a *semantic context* was first introduced by Chandy [**?**] in 1996 in the domain of C4I applications, primarily for military use and crisis response management systems[2]. Then, the *semantic context* of a user or application was called an *infosphere.*

An *infosphere* is the set of information pertaining to an agent—a person, computer system, company, piece of software, etc.—any entity that interacts and reacts to external events. Information constitutes an infosphere, and its components are data and processes, that is, both passive and active elements.

[2] C4I stands for “Command and Control, Communications, Computers, and Intelligence”.

An infosphere is contextual since the focus of the information is the center of the infosphere: its owner, the agent.

Within a military C4I context agents are, e.g., a general, a battleship, or a tank. Their infospheres share some characteristics: position, disposition, communication, etc. Other components are very different; the agents with which a general interacts are not the same as those with which a battleship or tank interacts.

There are a variety of proposals for specifying the contents of an infosphere. Most are fairly ad hoc, such as text and XML documents, and thus have little-to-no formal semantics. Consequently, there are few means by which one can formally reason about an infosphere.

Since a goal of this work is to specify the meaning and use of an infosphere within a formal system, it seems reasonable to (a) propose a mathematical model for an infosphere, and (b) call this model something new, to differentiate it from its informal cousin.

The formal model of an infosphere is dubbed a *semantic context* because, as discussed earlier, it is *contextual*, and, since its formal nature is being emphasized, it has a *semantics*. The theory of semantic contexts is specified with a formal method called kind theory [11,12,13].

1.1.1 Kind Theory

Kind theory is used to specify the semantics of reusable artifacts in collaborative environments. In the context of this work, the artifacts are the subcomponents of proof scripts and proofs, which in turn are instances of mathematical notions like variables and functions. The kind theoretical formalization of the semantic context, and likewise the UI features discussed here, follows partially from the grammar of the proof script language.

Kind theory is used because the process of writing a mathematical specification and proving properties about it are nearly wholly and exercise in knowledge reuse. For example, importing pre-existing definitions and theorems written and proven by other users is an action of knowledge reuse. Since kind theory's focus is on specifying and reasoning about reusable artifacts in collaborations, it is uniquely suited to this task.

The formalization of semantic contexts relies on only three basic operators of kind theory. The initial semantics of these operators is quite simple and can be interpreted in terms of set theory and sequences. So as to avoid immersing the reader in a new, unfamiliar formalism, these more familiar domains will be used in this paper.

The actual semantics are specified with kind theory, rather than these simpler foundations, because it provides a foundation for the future work discussed in Section 4.3. The reader interested in understanding the richer semantics should see the new PVS release and the aforementioned thesis [12].

1.2 Realizing the Semantic Context in PVS

To realize this new concept within a modern prover so as to provide a testbed for UI design, the PVS UI has been augmented: a built-in lexer and parser for the full PVS language have been added to the Emacs-based front-end. This infrastructure represents high-level PVS constructs such as declarations, theories, types, and proofs in a generic, syntactic fashion that mirrors the kind theoretical-structure of the proof script. This generic representation is used by a wide range of tools previously available only to "traditional" programmers using modern IDEs, and we aggresively reuse these tools in the new PVS UI.

The new functionality available through this parser-based approach includes construct browsers, quick access to construct definitions via a short-cut sidebar, menus, or implicit hyperlinks, contextual help, and context- and type-aware completion.

Other new non-parser-centric functionality has also been added to the PVS UI as well as the prover itself. First, the ability to visually expand and collapse structured elements of specifications and sequents has been added. Second, language elements can be represented graphically rather than textually, but with little impact on the ability to cut-and-paste terms. Finally, a user-extensible, type-aware pretty-printer has been added to the PVS prover.

1.3 Related Work

Other environments provide some of the features that are discussed in this paper. No environment has all of these features, and rarely is a formal foundation developed, nor do any use an integrated parser or the prover's proof script comprehension infrastructure.

1.3.1 Mathematics Environments

Some mathematics environments provide interactive UIs. Most theorem proving environments come from a "bare-bones" tradition. More effort is obviously spent focusing on mathematical, rather than visual, infrastructure. Until recently, with the growing adoption of Proof General [4], interfaces were nothing more than glorified command-lines [21].

Proof General and PVS are regarded as the most widely used environments with the most advanced UIs. Both use Emacs as a front-end, so much of their power is simply a side-effect of that choice. Features like syntax highlighting, completion, and hypertext documentation are examples of such pleasant side-effects. Neither Proof General nor, obviously until now, PVS has the features discussed in the following.

Some theorem proving environments like Jape and CtCoq have been used as UI/HCI testbeds [6,7,21]. Many of the innovations that were originally introduced in specialized environments have now found their way into general purpose front-ends. For example, proof-by-pointing in Centaur and CtCoq is now available in ProofGeneral, and the PPML-based layout and pretty-printing facilities of CtCoq are partially functionally reproduced in some of this work.

Pcoq is a Java-based user-interface for Coq [3]. It has some of the features of this work including a rich graphical interface and structured editing and presentation mechanisms. In particular, the rich presentation capabilities are based upon an internal formal representation of proof constructs. Unfortunately, while Pcoq followed a high-minded model put forth in [25], it has not evolved as is not used as a UI for Coq.

IsaWin is another environment which has some advanced interactive features [20]. IsaWin differs from this work in that it focuses on an iconic, graphical representation of mathematical constructs with a drag-and-drop interactive metaphor. As with most iconic languages, there are some problematic issues with scaling graphical representations of non-trivial artifacts.

Some commercial general-purpose computation environments, particularly quality commercial environments like Mathematica, Maple, and Matlab, have rich UIs. These tools provide WYSIWYG-ish interfaces that use mathematics fonts and provide direct editing of terms with a point-and-click, drag-and-drop UI. While the intention here is not to attempt to duplicate such UIs, their visual features are compelling and inspirational, but their editing features and flexibility are actually inferior to most sophisticated environments like those mentioned above.

Coupling an advanced UI like that of Mathematica to a prover like PVS is an interesting experiment [1]. But, as neither tool was designed as a reusable component, such coupling is awkward. Instead, in this work these commercial mathematics environments are studied, and the features that are found to be most useful in day-to-day work are adopted, adapted, and formalized for theorem provers.

1.3.2 Programming Environments

Modern programming environments, contrary to theorem proving environments, provide a large, rich set of UI features[3]. Common features that are not specific to traditional, non-mathematical programming include code browsing, automatic code and documentation formatting, type-aware and template-based completion, general project management, and integrated help, API documentation, process tracking, and support for revision control systems.

The major meta-claim of this paper is that each new IDE UI feature, after it has seen moderate success in the mainstream programming community, should be evaluated as a potential addition to theorem proving UIs. This work is the initial result of such an evaluation, focusing primarily on features that help users deal with the enormous proof environments mentioned in the introduction.

The next section describes the formal model of semantic contexts in detail. The sequel focuses on the UI elements that have been widely adopted in IDEs and have shown to be quite useful for dealing with the complexity problems of large-scale programming. The balance of the paper discusses how PVS has been extended to test out these ideas and summarizes some research opportunities in theorem prover UI design.

2 Theoretical Foundations

Kind theory has six basic operators: inheritance, inclusion, composition, interpretation, canonicalization, and realization, but we only have need of three of them for this work.

2.1 Basic Operators

Inheritance, written "<" and read as "is-a", is a reflexive, transitive, asymmetric relation between classifiers (eponymously called *kinds*) or instances of those classifiers at the object level. The logical rules involving inheritance state that (a) all *artifacts* (classifiers or objects) have a parent and, (b) if an inheritance relation exists between two artifacts, a description of how they

[3] Some of the tools that are evaluated, and from which ideas are borrowed, include Borland's jBuilder and Together ControlCenter, Eclipse, Eiffel Studio, IntelliJ IDEA, jEdit, Metrowerks CodeWarrior, NetBeans, Oracle9i JDeveloper, Sun ONE Studio, and the various "Visual*" tools (i.e., Microsoft's Visual Studio, IBM's VisualAge, WebGain's Visual Café).

differ exists as well[4].

Inclusion, written "$\supset$" and read as "has-a", is a containment relation, thus is also a reflexive, transitive, asymmetric relation between artifacts, that is, pairs of kinds or pairs of objects. The rules involving inclusion state that substructures of classifiers are preserved by substructures of classified objects. More specifically, if a kind K has a substructure kind L (written "$K \supset L$"), then an instance I : K has a substructure instance J : L ("$I \supset J$").

Finally, composition comes in two basic forms: the ability to break down an artifact into its subcomponents, written "$\otimes$", and the converse operator for creating a new artifact from existing subcomponents, which is written as "$\oplus$". Various logical rules relate the composition relations to other relations, particularly inheritance and inclusion, but those rules are not important for this discussion.

For the purpose of this work, one can think of of inheritance as a finite lattice of type-like structures, inclusion as set theoretic inclusion, and composition as textual concatenation (i.e., a document is a sequence of textual sub-elements composed through catenation).

2.2 Structure via Grammar

To formally represent the semantic context of an IME, its proof script language(s) must be classified with kind theory. The set of all classifiers identified in this section are called the *kind context* of the IME.

Since most formal languages have a (E)BNF description, a simple algorithm is available to help perform the basic classification steps:

- Canonicalize the BNF by flattening all choice-less rules that are typically used to clarify the grammar for the human reader.

  For example, rules of the form

  ```
  Rule1 ::= Rule2
  Rule2 ::= ...etc...
  ```

  should be flattened to

  ```
  Rule1 ::= ...etc...
  ```

- Identify each rule that defines a specific, independent concept within the proof script language. Create a new kind for each such concept, and let the unique rule define each kind. More is said below about these definitions.

  For example, the top-level concepts of PVS, as read directly from the PVS proof language's BNF, as well as emphasized in the PVS Language

[4] This description is not important for this work, but in short, it is two (total) functions: one that converts the parent to the child that is logically sound and complete, and another that converts the child into the parent that is forgetful, but sound and complete with respect to the parent's context.

Guide, are THEORIES and DATATYPES.

The second-level concepts of PVS are one level deeper within the grammar. They are exactly those concepts which are used to define the top-level concepts. They are: theory formals, import and export relations between theories, assumptions, declarations, judgments, and conversions.

If a construct is seen at multiple levels, the top-most level is the appropriate location for classification. For example, datatypes can be defined "inline" in PVS theories, but that does not preclude them from being a core top-level concept.

No relation exists between these identified classifiers as of yet.

- Now, identify all inheritance relations by picking out BNF rules that exhibit classification patterns[5]. Such patterns fall in two major forms in BNF, disjunctive choice and name similarity. Disjunctive choice indicates that a parent kind is the disjunctive composition (the "$\otimes$" relation) of its children kind.

  For example, declarations in PVS fall into nine subcategories, as emphasized by the grammar and the language reference: libraries, abbreviations, types, variables, constants, functions, formulas, and fields. Thus, these nine kinds are all subkinds of the kind representing declarations, and the declaration parent is decomposable into exactly one of its children kind.

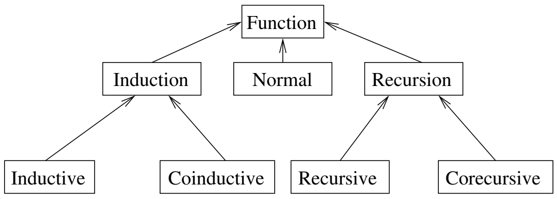


Fig. 1. The PVS Function Is-A Hierarchy

PVS functions fall into three subcategories: inductive, recursive, and neither inductive nor recursive, the first two of which are not mutually exclusive as a function can be both inductive and recursive. Induction and recursion each break down into two categories as well, as the dual notions of coinductive and corecursive are available. Thus, these notions define eight kinds in a two level classification hierarchy, as sketched out in Figure 1.

Higher-level notions are sometimes defined in terms of lower-level notions using composition. Thus, INDUCTION $\equiv$ INDUCTIVE $\oplus$ COINDUCTIVE means that an induction function is either inductive or coinductive, but not both. Likewise, FUNCTION $\equiv$ (INDUCTION $\otimes$ RECURSION) $\oplus$ NORMAL

[5] This is the basic procedure used to define an ontology or metamodel for a domain. Thus, if the language in question already has either of these defined, simple adopt that definition.

states that functions are either "normal", or exhibit some combination of inductive and recursion.

- To promote a classic, mathematically generic viewpoint on semantic contexts, that is, divorced from the terminology of the proof script language in question, all core constructs should also be classified using the standard mathematical concepts described in [12]. This permits users who are unfamiliar with the vernacular of an IME to still use its classification-based preference system, as described below.
- Next, identify substructure relationships in the proof script language by looking for BNF rules that exhibit containment relations. Typically, top-level concepts have single flattened rules that clearly indicate substructure.

  For example, the rule in the PVS BNF that describes the top-level concept of *theory* is

  ```
  Theory ::= id [TheoryFormals] ':' 'THEORY'
                [Exporting]
                'BEGIN'
                [AssumingPart]
                [TheoryPart]
                'END' id
  ```

  This rule clearly indicates that a theory kind has five substructures: a name (the *id* label), a theory formal, an exporting clause, an assuming part, and a theory part. Note that the names of these sub-rules even indicate their structural relationship ("Part").

  Analyzing the grammar from this point of view will produce an inclusion hierarchy for the entire language. This hierarchy will be used for several features discussed later in this document and is thus a key feature of the semantic context.
- Finally, identify compositional substructures by examining the mandatory textual ordering of the BNF rules.

  The aforementioned rule for PVS theories shows such a mandatory ordering: exporting clauses must precede assumptions which must in turn precede the theory part. Thus, a theory is defined as a sequence

  $$\textsc{TheoryName}, \textsc{TheoryFormal}, \textsc{ExportingClause}, \textsc{AssumingPart}, \textsc{TheoryPart}$$

  where the comma operator (",") is textual concatenation[6].

  Literal sequences identified in the grammar are also examples of compositional substructures. For example, the PVS BNF rule

  ```
  TheoryNames ::= TheoryName++','
  ```

  indicates that the classifier TheoryNames is composed of a comma-separated

[6] The comma operator is defined kind theoretically as a composition operator, since sequences can be constructed and deconstructed in the obvious way.

sequence of TheoryNames.

These sequences are also used for many features discussed later in this document.

2.3 Preferences by Classification

The classification hierarchies defined in the last section are primarily used by the UI to define semantic context-based preference settings.

Each concept can be *enabled* or *disabled* on a per UI feature basis. If a concept is *disabled*, then all of its subconcepts which are concepts below it in the is-a hierarchy are also *disabled*.

Using such a scheme, a user can intuitively describe to the system which concepts are appropriate for each UI interface, but unknowingly do so in an formal manner. For example, if a PVS proof script document is to be summarized in a tree view, and the user is only interested in the assumptions made in the theories, then the Assumption construct is *enabled*, and all other sibling concepts within the is-a hierarchy are *disabled*.

2.4 Iteration by Containment and Composition

The inclusion/containment and composition/sequence relations are generally used for iteration in the various UI features discussed over the next several pages.

A *memoizing reversed nested iteration* function is the key operator used in many UI features. Essentially, one must follow the sequence of instances within a context backwards and, each time one reaches the end of the sequence, move up to the enclosing context, but never return the same instance twice.

Consider this fragment of PVS code:

```
exists1 [T: TYPE]: THEORY
 BEGIN
  x, y: VAR T
  p, q: VAR pred[T]

  unique?(p): bool = FORALL x, y: p(x) AND p(y) IMPLIES (*)
...
```

Assume the user's focus, as discussed in the next section, is at the location labeled `(*)`. What is the sequence of reversed nested next objects?

The sequence is: $\ll$ `y`, `x`, `bool`, `p`, `unique?`, `q`, `T`, `exists1` $\gg$.

The first element is `y` because it is the final term in the enclosing sequence for the Forall quantifier instance. The variable `x` follows because it is the preceding instance at the same level. At that point, the sequence of the current level expires, so we move up one level to the constant declaration instance. Only three instances are in the sequenced substructure at this level:

$\ll$ `unique?`, `p`, `bool` $\gg$. They are iterated in reverse. When that sequence expires, the next enclosing sequence is followed: $\ll$ `x`, `y`, `p`, `q` $\gg$, but since `x`, `y`, and `p` have all already been chosen, only `q` can be returned. Finally, the elements of the outermost enclosing sequence $\ll$ `exists1`, `T` $\gg$ are returned in reversed order.

This generic iteration is legitimate in the PVS context because PVS is a higher-order prover, thus higher-order structures like formulas and theories can be directly referenced. For non-higher-order provers, the initial set of enabled classifiers will likely be smaller.

Features discussed later in this paper further refine this iteration operator. For example, type-aware completion uses this iteration to choose prospective terms to present for completion, but only shows each choice to the user if the term typechecks in the given context.

2.5 Identifying the Semantic Context of the User

The semantic context of the user is the composition of the underlying kind context as identified by the process of the preceding section, the instances of all associated documents, as determined by parsing the documents and identifying the concepts associated with all instances, and location of the current focus of the user.

The set consisting of all kinds identified using the process of Section 2.2 is the underlying *kind context* of the IME. This set is fixed for a given proof script language and IME combination, thus is not contextual from the point-of-view of a user.

All documents loaded by a user in the IME must be parsed to identify all instances in the current semantic context. A parser will obviously exist in the back-end prover of the IME, but it is often the case that no published API exists for accessing this information. Thus, many IMEs will use a front-end parser of some kind, like that discussed in Section 4.

Each construct identified by the parser has a starting point and an ending point. This *extent* information is used to determine the current user focus by identifying which construct encloses the current point of focus within the IME. If the current point does not fall within the extent of any instance, than the closest instance, moving backward through the current document, is consider the enclosing instance.

In total then, the semantic context of the user is the union of the fixed context of the prover and a dynamic context having two parts: (1) the contents of all proof scripts currently in use, and (2) the current focus of the user's attention.

In the next section various UI facilities are discussed that take advantage of the semantic context.

3 Semantic Context-based UI Facilities

Over half a dozen new facilities have been added to PVS that leverage the formalization of the user's context. While these new facilities primarily are of benefit to PVS users struggling with very large formalizations and proofs, we find that they also have significant utility to everyday PVS users.

3.1 Construct Browsing

A primary interface used by programmers in large-scale development efforts is some kind of feature browser. A browser lets a user quickly navigate a structured information space. There are a variety of UI alternatives for presenting hierarchical, structured information, the most popular of which are menus, trees, and panes. Each of these alternatives renders the hierarchical inclusion relation "$\supset$" of the user's semantic context.

3.1.1 Menu-based Browsing

Menu-based browsing is a popular way to navigate small-to-medium sized structured information spaces. The limitations of such an approach, e.g., the existence of many nested levels, many items at each level, etc., are well-known and often-abused. Thus, a menu's characterization, particularly its maximum size and depth, is user-tunable.

The PVS user can now specify with Emacs's *customize* feature which PVS constructs are shown in a summary menu, according to their classifying kind. By default, all constructs are shown in a hierarchical menu, organized by theory.

3.1.2 Hierarchical Shortcuts

Hierarchical shortcuts, often rendered as a tree, are a second popular way to present structured information. Filesystem explorers, like that found in Windows and OS X are examples of such tools, and IDEs like VisualStudio and Apple's Xcode organize project information in such a manner as well.

Hierarchies can be presented in either an integrated or independent (from the main display) fashion. When a construct in the summary tree is activated with a mouse click or keypress, the cursor in the current buffer jumps to the corresponding element in the main window. The user can customize which PVS constructs are summarized in the hierarchy on a classifier-centric basis.

3.2 Finding Definitions

Extended implicit hyperlinking is a feature new to commercial development environments. For example, automated spell-checking functionality, like that found in Emacs or Microsoft Word, is an example of implicit hyperlinking. In such an environment, the source document, perhaps a research paper, is implicitly hyperlinked to one or more dictionaries. The purpose of this particular hyperlinking is word spelling correction and definition and synonym lookup.

In general, the actions of an implicit hyperlink can either be informational, such as the definition of an operator or variable, or corrective, e.g., correcting the spelling of a variable.

Prior to this new work, PVS supported several kinds of implicit hyperlinks. For example, in PVS one uses a shifted middle mouse button click to show the type declaration of a construct is shown in a small information window. Unfortunately, most advanced features, like finding where a declaration is used, are not available as implicit hyperlinks and thus require the user to type long, hard-to-remember key sequences.

This functionality has been improved by increasing the number of recognized implicit hyperlink types. PVS keywords can now be activated to show their full definition related and usage information. Also, a variety of implicit link types can now be used in PVS comments including URLs, ISBN numbers, embedded image references, RFC titles, documentation cross-references, mail addresses, various compiler messages, pathnames, outline nodes, man pages, key sequences, table of contents entry, tag location, bibliography references, and some other more esoteric types.

Implicit hyperlinks of several kinds are particularly useful for theorem provers: URLs, embedded images references, documentation cross-references, and key sequences. URLs are obviously useful to cite relevant papers or other source material for the documented construct. Embedded images can be used to show pretty-printed versions of specific terms, sequents, or proof structures. Info nodes can be used to cross-reference other Emacs and PVS documentation, in particular the PVS release notes, which are the only part of the PVS documentation that at this time uses Texinfo. Finally, embedded key sequences can be used to document PVS UI behavior and trigger PVS actions so that, for example, the PVS tutorial can be made more interactive and automated.

3.3 Getting Help

Contextual help is generated by examining the current cursor position's enclosing construct and showing, after some user-tunable delay (usually a few hundred milliseconds), extra information about that construct.

Within PVS the help message is shown in a small area at the bottom of the user interface. Other environments often have similar unobtrusive-but-always-visible places in which put such assistance information.

Within the new PVS this functionality is used to document the grammar of language constructs (a kind of context-aware `help-pvs-prover-commands`) as well as the usage of prelude operators like boolean operators, conditionals, etc.

3.4 Completion

Modern programming environments offer scope- and type-aware completion. These IDEs parse and typecheck the input file as the user edits. When completion of a construct (e.g., a type, variable, or method name) is requested, a list of all legitimate (type-correct and visible within the current scope) alternatives is shown.

Unfortunately, because of PVS's heavy use of overloading and parameterization, this level of type-aware completion is not yet available in the new PVS UI[7]. At this time, when a completion is requested, only those constructs that are visible in the current scope and are *potentially* type-correct are shown. Completion choices are shown in the scope-dependent order selected by the algorithm of Section 2.4, with matching declarations in inner scopes shown before those in outer scopes.

3.5 Information Hiding

Outlines are used to show and hide various substructures of a document. An outline mode has been designed for the PVS UI, where substructures are identified based upon a document's inclusion hierarchy.

Structures are hidden and shown via modified mouse actions, like those used in documentation lookup, or via key sequences. A hidden structure is indicated with an elided hypertext region (which automatically highlights when touched by the mouse pointer) containing four periods. This representation was chosen so that it is similar to the elision that PVS pretty-printing performs today, but different enough that it is clear the elision is due to outline

[7] Currently the semantic package-based parser front-end is not communicating with PVS's typechecker, but this feature will be added in the future.

mode and not PVS itself. Touching a hidden structure with the mouse pointer for an extended period of time will temporarily un-hide the structure for examination.

Incremental searches on a buffer with hidden regions finds matches in hidden text, and such matches are made temporarily visible. If the user exits the search within such a temporarily hidden region, the text remains visible.

3.6 Pretty-Printing in PVS

A user-extensible, type-aware pretty-printer has also been added to the PVS UI.

The PVS input syntax is currently limited to ASCII characters. This is fairly restrictive, but PVS allows operators to be overloaded, using the type system and theory hierarchy to determine the operator in any given context. Overloading is very convenient, and heavily used in mathematics as the context usually makes clear which operator is meant. For example, the PVS prelude has four declarations for the caret operator (“`^`”) alone.

Pretty-printing the concrete PVS syntax is not difficult. Pretty-printing in PVS is handled using the Common Lisp Pretty Printing facility [24, Chapter 27] with a set of methods that walk down the PVS abstract term structures, which are implemented in CLOS.

In extended character sets like those available in LaTeXand UNICODE there are many more operator symbols available than in ASCII. When such character sets are available, it is desirable to map ASCII PVS operators to output operators of the alternative character set. For example, one might wish to use e^2 for `e^2` (exponentiation), and B_m^n for `B^(m, n)` (bit vector extraction). However, to properly translate such forms, more information than syntax is needed (in particular, types and resolutions).

The PVS LaTeX printer is driven from typechecked specifications. It allows substitutions to be made, and, by using the resolution information, can distinguish operators based on arity and the theory where the operator is declared. Types are not currently used, in order to keep the substitutions file simple[8].

Other approaches are available for pretty-printing that are especially useful when embedding different logics in PVS. For example, Skakkebaek [23] generated a new parser for the Duration Calculus, mapping its abstract syntax to subclasses of the PVS abstract syntax. Instances of these classes are pretty-printed using specialized methods, but typechecking, proving, etc. would use the methods of the superclass. This works well, but it is not easy to define a

[8] In order to properly provide support for types, parts of the substitutions file would need to be typechecked, and it is not trivial to determine the typechecking context.

new grammar in PVS[9]. Defining a new grammar for the Duration Calculus was not too difficult because it copied most of the PVS grammar and simply added a few new operators. Another difficulty is that as a proof is developed, terms get introduced that are part PVS and part Duration Calculus, and though they would be pretty-printed, the result could be confusing to the user.

In Pombo [22], PVS was used to provide the semantics of $\mathbf{A_g}$ specifications, defining the semantics of First Order Dynamic Logic and Fork Algebras, along with rules and strategies that allow a user to reason in $\mathbf{A_g}$. Here there were conversions defined, such as a meaning function, and arguments such as the current world of the Kripke structure, that by default are included in the prover interaction, but add clutter to the proof. In this case the function for pretty-printing applications was modified in order to suppress the meaning function and the world argument.

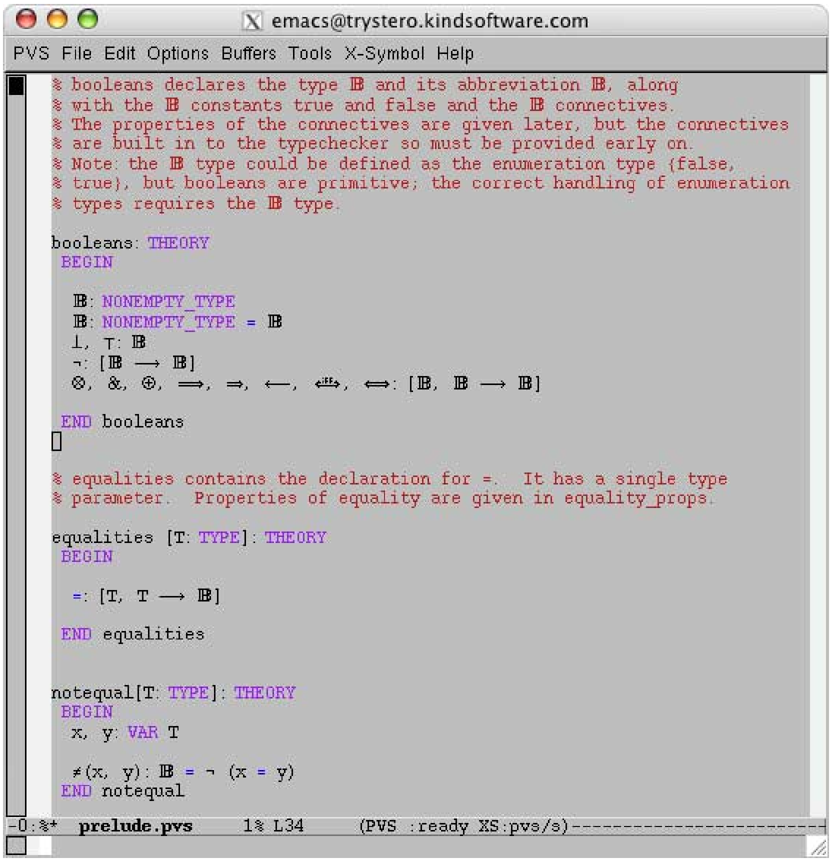
The new PVS pretty-printer is an elegent redesign that can be used to accomplish all of the above, but in a much simpler, more coherent fashion. The pretty-printer now that lets a user register pretty-print functions on a *per-type* basis. Each function takes a term and the type of the term as parameters. A function checks if the term given is one that requires special pretty-printing treatment, and pretty-prints the term if appropriate. If none of the registered functions are applicable, then the default pretty-printer is used. It is expected that some general functions will be provided that make it easy to perform common tasks, like suppress arguments and recognize applications whose operator is a constant of a specified theory and type. In its full generality, these functions act as semantic attachments for the customized pretty-printing of terms of any form.

This approach works well for many situations, but not all pretty-printing tools can be integrated into Common Lisp. For non-Common Lisp-based tool, resolution and type information need be provided. The most promising approach to this problem is to provide the abstract syntax of PVS terms in XML, given most programming languages have facilities for reading and manipulating XML documents. This is done, for example, in OMDOC, which provides an extension for PVS that generates OMDOC abstract syntax from typechecked PVS specs [14]. There are future plans for generating XML that directly reflect the internal abstract syntax of PVS, and this could easily be used to build a new pretty-printer.

[9] PVS currently uses the Ergo Parser Generator [15], which has a number of quirks that make it difficult to use.

3.7 Graphical Representation

Finally, some users prefer their computational mathematical environment to (syntactically) closely resemble their own non-computational, pen-and-paper approach to doing mathematics.


emacs@trystero.kindsoftware.com
PVS File Edit Options Buffers Tools X-Symbol Help

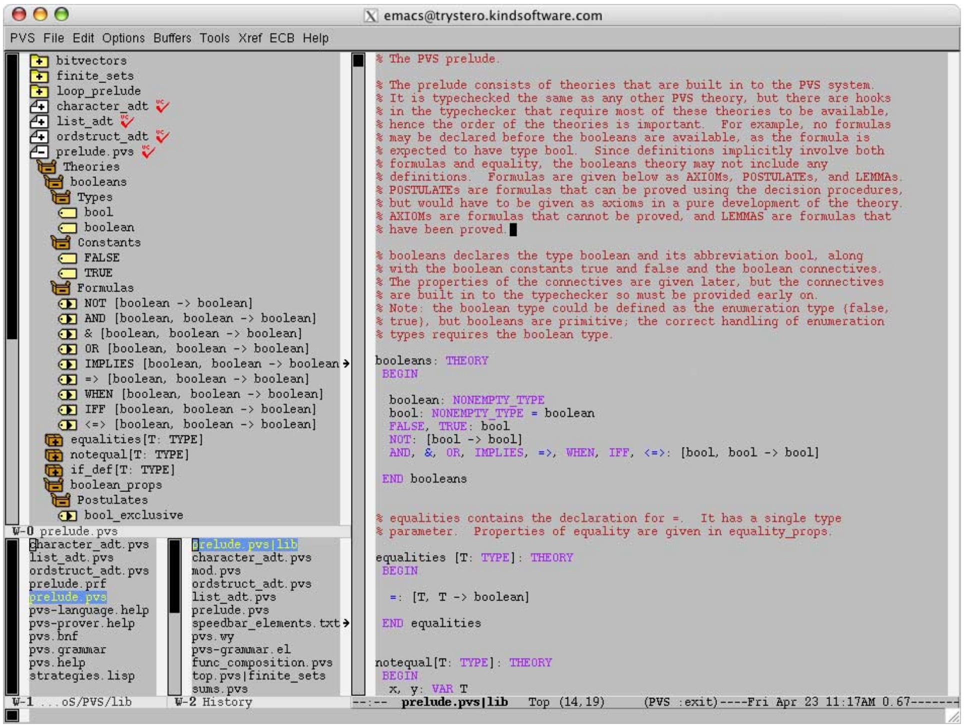
```
% booleans declares the type 𝔹 and its abbreviation 𝔹, along
% with the 𝔹 constants true and false and the 𝔹 connectives.
% The properties of the connectives are given later, but the connectives
% are built in to the typechecker so must be provided early on.
% Note: the 𝔹 type could be defined as the enumeration type (false,
% true), but booleans are primitive; the correct handling of enumeration
% types requires the 𝔹 type.

booleans: THEORY
 BEGIN

  𝔹: NONEMPTY_TYPE
  𝔹: NONEMPTY_TYPE = 𝔹
  ⊥, ⊤: 𝔹
  ¬: [𝔹 ⟶ 𝔹]
  ⊗, &, ⊕, ⟹, ⇒, ⟵, ⟸⟹, ⟺: [𝔹, 𝔹 ⟶ 𝔹]

 END booleans


% equalities contains the declaration for =.  It has a single type
% parameter.  Properties of equality are given in equality_props.

equalities [T: TYPE]: THEORY
 BEGIN

  =: [T, T ⟶ 𝔹]

 END equalities


notequal[T: TYPE]: THEORY
 BEGIN
  x, y: VAR T

  ≠(x, y): 𝔹 = ¬ (x = y)
 END notequal
-0:%*  prelude.pvs     1% L34      (PVS :ready XS:pvs/s)------------------------
```

Fig. 2. Using X-Symbol within the PVS UI

The PVS user interface is now capable of using extended character sets. Consequently, a standard set of mappings for all the core PVS operators is provided with our new PVS UI extensions. All uses of an overloaded operator look identical at this point in time. An piece of the PVS prelude, when rendered with symbols, is shown in Figure 2.

4 Extending PVS

While various new features of PVS have been mentioned in the past several sections, little detail has been given as to how the underlying functionality of the semantic context has been implemented, and how that functionality is used by the new UI. This section will briefly discuss the state of the current implementation.


emacs@trystero.kindsoftware.com
PVS File Edit Options Buffers Tools Xref ECB Help

```
bitvectors
finite_sets
loop_prelude
character_adt
list_adt
ordstruct_adt
prelude.pvs
 Theories
  booleans
   Types
    bool
    boolean
   Constants
    FALSE
    TRUE
   Formulas
    NOT [boolean -> boolean]
    AND [boolean, boolean -> boolean]
    & [boolean, boolean -> boolean]
    OR [boolean, boolean -> boolean]
    IMPLIES [boolean, boolean -> boolean
    => [boolean, boolean -> boolean]
    WHEN [boolean, boolean -> boolean]
    IFF [boolean, boolean -> boolean]
    <=> [boolean, boolean -> boolean]
  equalities[T: TYPE]
  notequal[T: TYPE]
  if_def[T: TYPE]
  boolean_props
   Postulates
    bool_exclusive
W-0 prelude.pvs
```

```
character_adt.pvs
list_adt.pvs
ordstruct_adt.pvs
prelude.prf
prelude.pvs
pvs-language.help
pvs-prover.help
pvs.bnf
pvs.grammar
pvs.help
strategies.lisp
W-1 ...oS/PVS/lib
```

```
prelude.pvs|lib
character_adt.pvs
mod.pvs
ordstruct_adt.pvs
list_adt.pvs
prelude.pvs
speedbar_elements.txt
pvs.wy
pvs-grammar.el
func_composition.pvs
top.pvs|finite_sets
sums.pvs
W-2 History
```

```
% The PVS prelude.

% The prelude consists of theories that are built in to the PVS system.
% It is typechecked the same as any other PVS theory, but there are hooks
% in the typechecker that require most of these theories to be available,
% hence the order of the theories is important.  For example, no formulas
% may be declared before the booleans are available, as the formula is
% expected to have type bool.  Since definitions implicitly involve both
% formulas and equality, the booleans theory may not include any
% definitions.  Formulas are given below as AXIOMs, POSTULATEs, and LEMMAs.
% POSTULATEs are formulas that can be proved using the decision procedures,
% but would have to be given as axioms in a pure development of the theory.
% AXIOMs are formulas that cannot be proved, and LEMMAS are formulas that
% have been proved.

% booleans declares the type boolean and its abbreviation bool, along
% with the boolean constants true and false and the boolean connectives.
% The properties of the connectives are given later, but the connectives
% are built in to the typechecker so must be provided early on.
% Note: the boolean type could be defined as the enumeration type {false,
% true}, but booleans are primitive; the correct handling of enumeration
% types requires the boolean type.

booleans: THEORY
 BEGIN

  boolean: NONEMPTY_TYPE
  bool: NONEMPTY_TYPE = boolean
  FALSE, TRUE: bool
  NOT: [bool -> bool]
  AND, &, OR, IMPLIES, =>, WHEN, IFF, <=>: [bool, bool -> bool]

 END booleans


% equalities contains the declaration for =.  It has a single type
% parameter.  Properties of equality are given in equality_props.

equalities [T: TYPE]: THEORY
 BEGIN

  =: [T, T -> boolean]

 END equalities


notequal[T: TYPE]: THEORY
 BEGIN
  x, y: VAR T
--:--  prelude.pvs|lib   Top (14,19)     (PVS :exit)----Fri Apr 23 11:17AM 0.67-------
```

Fig. 3. The Integrated PVS Environment

Figure 3 shows an example configuration of the new integrated environment [10]. As seen in the figure, the integrated PVS environment has several subcomponents. Shown here are a hierarchical tree-based layout of the current proof script (upper-left in window "W-0"), as discussed in Section 3.1.2, a list of all files in the currently selected folder (window "W-1"), and a "history" of the files that have been most recently visited (window "W-2"). The main window shows the current proof script that is being edited.

[10] The observant reader will note that the screenshots were made on a system running OS X. The reason for this choice is not simply aesthetic. The author is also contributing to a port of PVS to various Open Source Common Lisp implementations, thus we have PVS partially running on OS X at this time.

The configuration shown is the author's favorite layout, but other users will have different needs given their working methodology, screen size, etc. The various subwindows can be interactively arranged and resized, and their positions are automatically saved from session to session.

This integrated environment is an extension of the Emacs Code Browser (ECB), a source code browser for Emacs [5]. It displays a user-customizable set of windows that can be used to browse directories, files, and file contents, all of which are substructures of the current semantic context. The summary pane in the top-left of the future can also be split off into a separate window. The *Speedbar* package is used to support this functionality [19].

Shortcut menus (not seen in the illustration) are available as discussed in Section 3.1.1. They are realized using the *Imenu* facility, which is built into recent versions of Emacs. This features offers a way to find the major definitions in a file by name, via a nested set of menus.

Finally, the implicit hyperlinking functionality discussed in Section 3.2 is realized by reusing portions of the Hyperbole package [28].

4.1 Underlying Realization

The most complex part of this work is mapping the constructs found in the proof scripts to an underlying object-oriented representation.

The underlying representation of the kind theoretical model of the PVS language is realized using *eieio*, a CLOS implementation for Emacs lisp (elisp) [18]. The object-oriented structure of the model maps directly onto CLOS objects in a natural fashion.

To implement the mapping function, one needs to either (a) write PVS-specific functionality for Emacs or (b) rely upon a generic language-based framework that enables the use of existing and future tools based upon the framework. With an eye toward saving time, increasing reliability, and taking advantage of the hard work of other Emacs enthusiasts, option (b) was chosen. And, after evaluating the various generic framework's available for Emacs, the *semantic package*, a part of the *CEDET* framework, was chosen as the foundation library.

The semantic package for Emacs is used to build lexers and parsers in elisp [17]. To use the semantic package with a new language, thereby enabling all semantic package-based tools, one must write a grammar specification of the language. Essentially, by relying upon a generic foundation, PVS support is enabled "automagically" for all of the semantic package-dependent tools, discussed in this section including iMenu, speedbar, and the ECB.

The semantic package includes, at its core, a lexer generator, and a compiler compiler, or what is known as a *bovinator* in the semantic package's nomenclature. The core utility is the *semantic bovinator* which has similar capabilities to the GNU `bison` compiler compiler [8]. Recent versions of the semantic package (mid-2003) include a second compiler compiler called *wisent*, which is effectively a full GNU `bison` implementation in elisp. The work described in this paper initially used the bovine framework, but now uses wisent.

The final bit of functionality, that of rendering PVS proof scripts with extended character sets as seen in Figure 2, is accomplished with the X-Symbol package [27].

4.2 Future Improvements

There remain several new features and refinements that need to be added to the new PVS UI.

First, a graphical toolbar, like that available in Proof General has to be added to represent some of the new features. Some Emacs users will simply ignore new features until they have such toolbars, so it is important to add this simple feature soon.

Second, currently the wisent parser does not work with the X-Symbol package, as the grammar is written in ASCII. Thus, a user currently either can have symbolic representation of their PVS proof scripts, or can have the integrated environment, but not both. We hope to lift this limitation before the release of the new PVS.

Keeping track of the importing and exporting relationships between PVS theories is a complex and sometimes time-consuming task. A new semantic package that is under development called COGRE can graphically represent semantic package parsed structures [16] (initially, COGRE's focus is UML diagrams). In COGRE, a graphical representation can be manipulated to change the underlying associated data. Consequently, some experimentation with the graphical representation of theory relationships is warranted.

Finally, UI operations involving parameterized theories are weak. Improving support for completion and summarization of such parameterized substructures is an important research topic. It is likely that PVS theory parameterizations can be characterized by the kind theoretical composition operators, thus the methods of Section 2 can continue to be used.

4.3 Research Opportunities

Many currently popular IDEs have recently added refactoring functionality [10,26]. Refactoring is the process of changing the implementation of a system in order to improve some aspect of the implementation while preserving its capabilities. Examples of refactoring include safe renaming of program features (variables, functions, methods, etc.), function extraction and insertion, and safe modification of feature visibility (e.g., making a public variable private, automatically writing getter and setter methods, and automatically inserting these methods at all program locations that access the original public variable).

An example refactoring operation within an IME is the identification and use of a cut rule. When performing a large proof, it is sometimes the case that, at some mid-point during the interactive process, multiple branches of the proof are recognized as having a similar structure. To simplify the proof, a lemma could be defined and used within these branches. Unfortunately, many provers, PVS included, provide no means by which such a lemma, acting here as a cut, can be extracted, defined, proved, then (re)used.

Theoretically representing refactoring operations within an IME is a significant research opportunity, as it will provide the foundation for formal verification of refactoring operations.

If the semantics of a refactoring operation can be represented within kind theory as what is known as an *interpretation*, then the verification effort is trivial, as a fundamental property of full interpretations is that they are sound and complete, that is, they preserve all properties of the constructs being interpreted.

Because the mathematical structures, including the proof theoretic constructs (logical rules, judgments, etc.) of the IME are represented with kind theory, then meta-theoretical claims, like the soundness the proof context under interpretation (e.g., refactoring), are possible.

More work is necessary along these lines to determine if such a general formal framework like kind theory is appropriate and useful for IMEs, but the early results look promising.

Acknowledgments.

This work was supported by the Netherlands Organization for Scientific Research (NWO). Thanks to Sam Owre, Erik Poll, Martjin Warnier, Bart Jacobs, and Adriaan de Groot for their input on earlier versions of this paper. Thanks to Chara Williams for help with the screen captures.

References

[1] Andrew Adams, Martin Dunstan, Hanne Gottliebsen, Tom Kelsey, Ursula Martin, and Sam Owre. Computer algebra meets automated theorem proving: Integrating Maple and *PVS*. In Richard J. Boulton and Paul B. Jackson, editors, *Theorem Proving in Higher Order Logics,*

TPHOLs 2001, volume 2152 of *Lecture Notes in Computer Science*, pages 27–42, Edinburgh, Scotland, September 2001. Springer–Verlag.

[2] J. Alves-Foss, editor. *Formal Syntax and Semantics of Java*, volume 1523 of *Lecture Notes in Computer Science*. Springer–Verlag, 1999.

[3] Ahmed Amerkad, Yves Bertot, Loïc Pottier, and Laurence Rideau. Mathematics and proof presentation in Pcoq, 1998.

[4] David Aspinall. Proof General: A generic tool for proof development. In *Proceedings of Tools and Algorithms for the Construction and Analysis of Systems (TACAS'00)*, volume 1785 of *Lecture Notes in Computer Science*. Springer–Verlag, 2000.

[5] Klaus Berndl, Jesper Nordenberg, Kevin A. Burton, and Eric M. Ludlam. The ECB user manual, July 2003. See http://ecb.sourceforge.net/

[6] Janet Bertot and Yves Bertot. CtCoq: A system presentation. In *Algebraic Methodology and Software Technology*, pages 600–603, 1996.

[7] R. Bornat and B. Sufrin. Jape's quiet interface. In *Proceedings of User Interfaces for Theorem Provers (UITP'96)*, 1996.

[8] The GNU Foundation. The GNU bison manual, February 2002. http://www.gnu.org/software/bison/bison.html.

[9] Bart Jacobs and Erik Poll. A logic for the Java modeling language JML. Technical Report CSI-R0018, Computing Science Institute, University of Nijmegen, November 2000.

[10] JetBrains. IntelliJ IDEA 3.0 overview, 2002. http://www.intellij.com/

[11] Joseph R. Kiniry. A new construct for systems modeling and theory: The Kind . Technical Report CS-TR-98-14, Department of Computer Science, California Institute of Technology, October 1998.

[12] Joseph R. Kiniry. *Kind Theory*. PhD thesis, Department of Computer Science, California Institute of Technology, 2002.

[13] Joseph R. Kiniry. Using kind theory for distributed knowledge capture. In *Proceedings, DC-KCAP '03, Distributed and Collaborative Knowledge Capture Workshop at K-CAP '03*, 2003.

[14] Michael Kohlhase and Sam Owre. An OMDoc interface to *PVS*, 2001. http://www.mathweb.org/cvsweb/cvsweb.cgi/omdoc/projects/pvs/

[15] P. Lee, F. Pfenning, J. Reynolds, G. Rollins, and D. Scott. Research on semantically based program-design environments: The Ergo project in 1988. Technical Report CMU-CS-88-118, Department of Computer Science, Carnegie Mellon University, 1988.

[16] Eric Ludlam. The COGRE manual, 2002. http://cedet.sourceforge.net/cogre.shtml

[17] Eric Ludlam. The Semantic manual, 2002. http://cedet.sourceforge.net/semantic.shtml

[18] Eric Ludlam et al. The Eieio manual, 2003. http://cedet.sourceforge.net/eieio.shtml

[19] Eric Ludlam et al. The Speedbar manual, 2003. http://cedet.sourceforge.net/speedbar.shtml

[20] C. Lüth, Tej H, Kolyang, and B. Krieg-Brückner. TAS and IsaWin: Tools for transformational program developkment and theorem proving. In J.-P. Finance, editor, *Fundamental Approaches to Software Engineering FASE'99. Joint European Conferences on Theory and Practice of Software ETAPS'99*, number 1577 in Lecture Notes in Computer Science, pages 239–243. Springer–Verlag, 1999.

[21] N. Merriam and M. Harrison. What is wrong with GUIs for theorem provers? In *Proceedings of User Interfaces for Theorem Provers (UITP'97)*, 1997.

[22] Carlos López Pombo, Sam Owre, and Natarajan Shankar. A semantic embedding of the $\mathbf{A_g}$ dynamic logic in *PVS*. Technical Report SRI-CSL-02-04, Computer Science Laboratory, SRI International, 333 Ravenswood Ave., Menlo Park, CA 94025, July 2003. To appear.

[23] Jens U. Skakkebæk and N. Shankar. A Duration Calculus proof checker: Using *PVS* as a semantic framework. Technical Report SRI-CSL-93-10, Computer Science Laboratory, SRI International, 333 Ravenswood Ave., Menlo Park, CA 94025, December 1993.

[24] Guy Steele. *Common Lisp: The Language*. Digital Press, second edition, 1990.

[25] Laurent Thery, Yves Bertot, and Gilles Kahn. Real theorem provers deserve real user-interfaces. Technical Report 1684, Sophia Antipolis, May 1992.

[26] Marian Vittek. The Xrefactory system, 2002. http://www.xref-tech.com/

[27] Christoph Wedler. The X-Symbol manual, May 2003. http://x-symbol.sourceforge.net/

[28] Bob Weiner et al. BeOpen.com Hyperbole: The everyday net-centric information manager, July 1999. http://sourceforge.net/projects/hyperbole/