SOFTWARE VERIFICATION RESEARCH CENTRE

DEPARTMENT OF COMPUTER SCIENCE

THE UNIVERSITY OF QUEENSLAND

Queensland 4072
Australia

TECHNICAL REPORT

No. 92-8

Providing User-Oriented Support for Software Development by Formal Methods

A.S.K. Cheng, J. Han, J. Welsh and A. Wood

December 1992

Phone: +61 7 365 1003
Fax: +61 7 365 1533

Providing User-Oriented Support for Software Development by Formal Methods

Anthony S.K. Cheng, Jun Han, Jim Welsh and Andrew Wood
Software Verification Research Centre
Department of Computer Science
The University of Queensland, 4072, Australia

Abstract

Existing tools for software development by formal methods lack user-oriented properties necessary to their effective use in software engineering. We propose a strategy for overcoming these deficiencies by interposing a user-oriented front-end editor between the user and the formal method tool itself. We demonstrate the feasibility of this approach by adapting an existing generic language-based editor to provide a front-end to an existing theorem prover. To cater for the full range of structures which arise in formal methods, we propose to define a generic model of formal method documents and to develop a corresponding front-end editor for such documents which is capable of real-time interaction with a corresponding variety of tools supporting formal methods.

1 Introduction

Software development by formal methods must be seen as an advanced form of computer-aided software engineering. Development by such methods involves an overwhelming amount of technical and managerial detail in general, and semantics-based guidance in particular, which is only practical with computer-aided support. There have been many efforts to develop computer-based support tools for such software development. To date, however, most of them have concentrated on demonstrating the feasibility of the automated semantic support essential to formal methods while providing limited clerical and managerial assistance. Few of them provide systematic user-oriented support from the software engineering viewpoint.

User-oriented support is necessary if formal method tools are to be practical in industrial applications. This support implies both an easily usable physical interface and reinforcement of the user's conceptual model of software development in the methodology concerned.

Our research focuses on support for the user's conceptual model of software development by formal methods, with particular emphasis on easy construction, modification, review and reuse of software products and processes. In this paper, we report our work on developing a user-oriented support environment for existing formal method tools. Our objectives are as follows:

1. The approach should be potentially applicable to many formal method tools.
2. The changes to existing tools should be minimal.
3. The work should also provide insights into achieving user-oriented support in newly developed formal method tools.

After giving a brief analysis of existing formal method tools in section 2, we introduce an approach to providing user-oriented support environment for these tools in section 3. To demonstrate the practicality of the approach, in section 4 we present a case study of providing a user-oriented support environment for an interactive theorem prover. In section 5 we consider the problems to be solved in generalising the approach to a wider range of formal method tools.

2 Analysis of formal method tools

The development of a software system in a formal method involves a sequence of well-defined steps. The assumptions and results of each step have semantic relationships defined with mathematical rigour. The relationships can be checked formally, and the results may even be obtained by automatic calculation from the assumptions and the type of calculation implied by the given step.

Software development by formal methods involves two major tasks: program development and theorem proving. Over the last two decades, many prototype tools for program development and theorem proving have been developed to investigate computer-aided support for formal methods. These include: HDM [17], EHDM [16], Gypsy [9], Affirm [8, 12, 13], EVES [7], several assistants for the refinement calculus [4, 18, 2], Mural [11], RAISE [14], B [19], Demo2 [15], HOL [10], LF [1], and Nuprl [6]. Most of these tools concentrate on the semantic checking and calculation involved in the development steps. Limited attention is given to systematic support for the development process.

To capture the relationships between its components, there is a need to record the development process. In an engineering environment the design process is recorded so that one can relate the requirements to the product. In many existing formal method tools, the development process is regarded as insignificant once the solution is obtained. Typically, a tool is used to apply the formal method to the problem once. Consequently, recording the development process is regarded as non-essential. For example, the proof process is not systematically recorded in the theorem proving systems Demo2 and HOL. Without recording the development process, however, a tool cannot present the concepts inherent in the formal method clearly and the user cannot review or reuse an existing development.

Natural presentation of method concepts by support tools is critical in helping the user to apply a method. The absence of such support may cause inconsistency between the user's model of the method and the tool's presentation of the method. With this inconsistency, the user cannot utilise the tool efficiently because a continuous mental translation is needed between the user's conceptual model and the presentation. By not recording the development process, many existing formal method tools can only focus on the presentation of the current development step rather than (some abstraction of) the entire development which natural presentation may require. For instance, Nuprl's presentation is based on a proof step while HOL's presentation is based on the current formula to be proved.

In any non-trivial development, a review facility is essential to understanding the development process. There are two possible types of review. The user can review the development process statically, simply by browsing through appropriate recorded details. Alternatively, the system can animate the development process by replaying it for the user to observe. Most formal method tools provide very limited support for reviewing the development process. Many are developed based on ASCII terminals and some of these, such as EVES, do no more than allow the information scrolled off the screen within the current session to be recalled using special system commands.

In any software development, good documentation facilities are essential for the review process. This documentation assists the understanding of the rationale behind the development. Although the recorded commands show the actions taken at each step, they do not necessarily explain the reasons behind the step. Without additional documentation, it may be difficult for the reviewer to understand the intention of each action. Lack of documentation support is a problem suffered by almost all formal method tools.

As in any software development methodology, reuse of the intellectual effort invested in solving one software problem to solve another related problem is vital to the economic application of formal methods. Distinct problems solved by a given formal method may have similar properties. In principle, therefore, part of the development process for one problem may be reused for another problem. In a restricted sense of reusability, the user may simply use the results from previous problems to solve new problems. Many formal method tools, including Demo2, Mural and EVES, support reuse of results such as proven theorems. However, solving problems with similar properties by this approach can only achieve limited success since it depends on an appropriate structuring of the results produced in solving the first problem to enable their use in the second. In many cases, the existing results are inappropriate and a distinct but similar development process has to be recreated. In doing so reuse of the existing process may involve some combination of the following:

- The user replays the existing development up to a given point then interrupts it and continues in a distinctive fashion.
- The user copies a segment of an existing development to the new one.
- The user systematically edits specific commands and fields in an existing development to produce a structurally similar but semantically distinct result.

Since they don't capture the development process, most existing formal method tools inevitably cannot support such reuse by editing and replay.

3 An approach to providing user-oriented support

In principle, the deficiencies of existing formal method tools discussed in the preceding section can be corrected in two ways. One option is to extend the functionality of each tool to provide appropriate facilities for development capture, documentation, review and editing. For existing tools, however, such extension would be difficult if not impossible. The alternative and preferred option is to encapsulate the facilities for capture, documentation, review and editing in a generic front-end with which a new or existing formal method tool can be integrated (Figure 1). The purpose of the integration is to move most of the functionality of a user interface from the tool to the front-end and to provide improved support for the user's conceptual model of the development process without changing the tool significantly. Any user command for the tool will be transmitted by the front-end to the tool. Similarly, all the responses from the tool will be relayed back to the front-end. In this paper, we shall concentrate on the support to be provided by the generic front-end. The details for the integration between the front-end and the formal method tool will be reported in a separate paper [5].

All the problems identified in section 2 are problems common to a range of formal method tools. The broadly-based solution developed in this section can be applied to more than one tool. In this sense the front-end is considered to be generic, and its implementation can clearly exploit this genericity by enabling the integration requirement of a range of tools to be determined by appropriate integration parameters.

The front-end views the user and the formal method tool as working cooperatively to develop a solution to the given problem. This cooperation can be recorded in a document during the development. To achieve this, the front-end captures the following information in a document: any environmental information accessible and used during the development, and every user input and tool response, including final results. The front-end also captures the development relationships between elements of this information. These development relationships will provide information such as how the material has been used, and how a solution is obtained for the given problem. All this information together forms the basis of the document structure. In order to make the development process more understandable, the front-end will also allow the user to supply additional documentation, wherever it is needed, to produce a *literate document* [3].

Being able to record the necessary information is insufficient. Presentation of the document, both at the level of individual development steps and as an overall development structure, is also an important part of the strategy. At either level, the presentation must reflect the conceptual model of the formal method naturally. In some cases the development structure may be linear, requiring effective sequential presentation techniques. In other cases hierarchic structures will commonly arise, and be presented accordingly. Most generally, the development relationships may form a graph-like structure, to be presented either graphically or via relational (hypertext-like) navigation. In all cases, relevant presentation techniques must cater both for interactive review and manipulation of the document concerned and for the production of hard copy literate documents.

As with any document, editing is an integral part of the manipulation process to be supported in the front-end. To the user, editing is the mechanism which enables reuse of existing developments. The user can copy any existing development process to a new document and adjust parts of the copied section to suit the needs of the new problem. Thus, development processes for similar problems can be reused and solutions for new problems can be obtained effectively.

The architecture proposed, of a user-oriented front-end which enables document preparation, display and manipulation in communication with a semantics-oriented back-end tool, is by no means only appropriate to formal methods, and has been advocated as a general architecture for development tools elsewhere [21]. The distinctive requirement of formal methods, however, is that the document preparation process requires real-time semantic assistance which determines part of the document at each step and constrains the user's next options. In effect, such document preparation is semantics-directed, and the interface between the front-end and the formal method tool must support tool-directed input.

4 Front-end support for a theorem prover

The approach proposed above has been tested by prototyping front-end user-oriented support for an existing formal method tool. The chosen base for the prototype front-end is the generic language-based editor UQ1 while the theorem prover Demo2 is the existing formal method tool. Descriptions of both tools

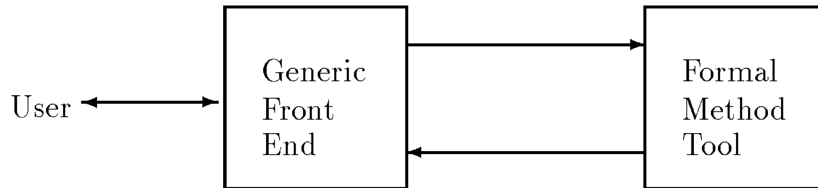

Figure 1: Relationship between generic front-end and formal method tool

and the reasons for choosing them are given before presenting the results from the case study.

4.1 The chosen front-end

The generic language-based editor UQ1 [20] is a recognition editor which supports direct manipulation of structured textual documents with embedded unstructured commentary or documentation. The editor parses user input in real time according the language-specific syntax description predefined by the editor builder. Document display is structure-based and exploits a distinguished nested context structure in the document. A presentation view focuses on a particular context and displays the document information within that context down to a prescribed nesting level. Multiple presentation views focusing on different contexts can be displayed at the same time. Being an editor, UQ1 supports all the standard editing facilities using a direct textual manipulation paradigm. UQ1 is also capable of generating a LaTeX document from the input and of integrating with analytical tools via a range of integration techniques [21].

UQ1 is chosen as the basis for a prototype front-end because it has most of the properties specified in the last section. The document supported by the editor can capture structured inputs from both the user and the tool, and include literate documentation. Hierarchical presentation structure is provided by the context mechanism. UQ1 can produce a printed document as well as having good interactive manipulation and editing facilities.

4.2 The chosen formal method tool

Demo2 [15] is an interactive theorem prover. Proof proceeds by a sequence of expression transformations, but includes the so-called window inference technique, where the user selects a subterm from an expression, transforms the subterm, substitutes the result back to obtain an expression equivalent to the original. The selection process is known as “window opening”. When a transformation window is opened, the original expression is put aside and the attention is switched to the selected subterm. All the transformations on the selected subterm occur inside the transformation window. On closure of this window, the transformed subterm is substituted back into the parent expression. The effect of a transformation window is similar to proving and using a lemma — the first expression can be transformed into the second with the intermediate justification proven in the window. Figure 2 shows a typical Demo2 proof session with commentary.

Demo2 is chosen for the case study because its user interface has the deficiencies typical of a range of formal method tools. The output is presented on the screen sequentially and shows little correspondence to the hierarchical structure of the transformation window concept. Demo2 logs all the commands used in a session primarily as a precaution against system changes. This command log can be “replayed” to restore Demo2 to the final state reached, but no user control over this replay is available. The log does not separate the commands from different proofs within one session, or include a history of the tool's responses. As such it is a poor basis for review of an existing proof. Any information that is scrolled off the screen cannot be retrieved without a replay. Although proven theorems can be used to prove new theorems, their development processes cannot really be reused. In principle the user can edit and then replay a command log to produce a modified result, but has no benefit from Demo2's interactive guidance in doing so. During proof construction local editing is also primitive. The user can only “undo” the current transformation window, i.e., truncate the development process to date to the point where the current window was opened.

4.3 Application of the approach

In this section, we describe how an appropriately modified UQ1 supports the theorem proving activities of Demo2 and provides better manipulation facilities for Demo2 proofs. The document captured

Demo2 Session	Commentary
/fol: prove(doubleneg, (not (not #A)) <=> #A).	Begin the proof on the expression.
focus: not (not #A) <=> #A Equivalence: <=> Goal: true	Demo2's window state showing current expression, equivalence being employed, and goal to achieve.
/fol (window_1,theorem - top level): trans(def(<=>)).	Expand the definition of “<=>” using “=>” and “and”.
focus: (not (not #A) => #A) and (#A => not (not #A)) Equivalence: <=> Goal: true	
/fol (window_1,theorem - top level): openwin([1,1,1]).	Concentrate on the 1st conjunct. Open a window on the 1st “not #A”.
focus: not #A 2: not #A 1: #A => not (not #A) Equivalence: <=> Goal: true	Hypotheses deduced by Demo2 from the previous focus. Note that hypothesis 2 is an immediate match for the focus.
/fol (window_2,theorem): trans(bool, [2]).	Transform using boolean introduction.
focus: true 2: not #A 1: #A => not (not #A) Equivalence: <=> Goal: true	Focus and the hypothesis agree and result in “true”.
/fol (window_2,theorem): closewin.	Return the result.
focus: #A => not (not #A) Equivalence: <=> Goal: true	Substitute the result and simplify the expression — the first conjunct is proven.
/fol (window_1,theorem - top level): openwin([2,1,1]).	Now concentrate on the 2nd conjunct. Open a window on the 2nd #A.
focus: #A 1: #A Equivalence: <=> Goal: true	The deduced hypothesis again offers an immediate solution.
/fol (window_2,theorem): trans(bool, [1]).	Transform using boolean introduction.
focus: true 1: #A Equivalence: <=> Goal: true	Focus and hypothesis agree and result in “true”.
/fol (window_2,theorem): closewin.	Return the result.
focus: true Equivalence: <=> Goal: true	The expression is proven to be true.
/fol (window_1,theorem - top level): closeproof.	Complete the proof.

Figure 2: A typical Demo2 proof session

by the UQ1 front-end is the interleaved sequence of user commands and Demo2 responses. Since the document is constructed cooperatively between the user and Demo2, a real-time communication channel between UQ1 and Demo2 must be established. UQ1's existing integration mechanisms use such channels but do not allow document fragments to be contributed by the tool as well as the user. A number of modifications were therefore made to UQ1 and its document description language to accomplish the necessary communication during input. The details of these modifications are explained in [5]. With these changes, capture of the user-Demo2 dialogue as a UQ1 document was easily achieved.

UQ1's existing presentation facilities immediately enabled natural presentation of this document. As pointed out in the previous section, each transformation window in Demo2 shows the transformation sequence performed on an expression. Any transformation of a subterm within the expression is performed in another nested window. The transformation windows from Demo2 thus map naturally onto the UQ1 context structure. Both use hierarchical structure to separate more detailed information from information at the enclosing level. By exploiting this UQ1 facility, the sequence of transformations applied to a given Demo2 expression can be seen within a single context. Those applied to a subterm or lemma proof are viewed by zooming into a nested context.

Figure 3 shows how UQ1 presents the example given in Figure 2. The upper display window shows the overall proof sequence while the lower display window shows the transformation sequence within the transformation window whose **openwin** is highlighted in the upper display.

Improved presentation is also achieved using other UQ1 display capabilities. These include displaying ASCII input symbols in more familiar graphic representations such as presenting `not` as ¬, and enhancing the display of the window information by construct boxing. Literate document support is provided by allowing comments between the prompt ":" and the command. The user can describe the rationale of the development using this comment facility. The comments are only seen by UQ1, and are invisible to Demo2. Note, however, that comments may include LaTeX commands which affect their format when a hard copy document is produced.

With the Demo2 development process captured as a UQ1 document in this way the user can browse through the document, and hence review the development, using all of UQ1's browsing facilities. These include zooming and panning between related contexts, relocation via a context menu, and searching for specific names, fragments or patterns. Likewise the user can invoke UQ1's existing facility for the generation of a typesetable LaTeX document from the interactive representation. Appendix A shows the hard copy produced by LaTeX from the proof displayed in Figure 3. The detailed conventions for on-line presentation and for hard-copy document production are determined by generic inputs to UQ1 in association with the basic document description, and can thus be customised to individual user or organisation needs without any change to either UQ1 or Demo2.

Demo2 only supports sequential construction of proofs while UQ1 imposes no restriction on the editing sequence within the document. To maintain effective Demo2 support when the user applies an edit operation to the UQ1 document, synchronisation is needed to reconcile the different modes of operation in Demo2 and UQ1. The strategy adopted is to ensure that Demo2 is appropriately synchronised each time a Demo2 response is expected during input. During sequential input this will happen automatically but when a user insertion is attempted explicit resynchronisation may be required. The simplest means of achieving this is to reset Demo2 to the state appropriate to the start of the document and then "replay" each user command from Demo2's position to the current edit position. The form of replay used for synchronisation is of course a silent one which is unobservable to the user but for large proofs could still take a considerable time. In practice, the replay overhead can be reduced by using Demo2's undo-window facility to attain the first available position upstream from the edit point and then replay from there. In general, the synchronisation strategy available is limited by the capacity of the back-end tool to reset its state efficiently.

This provision of user editing of the UQ1 document with synchronised reprocessing by Demo2 enables modification or reuse of existing Demo2 development in a natural way. Although the user can make use of all the editing facilities provided by UQ1, in doing so, some restriction may be imposed on the area of editing. It is illogical to allow the user to edit any information generated by the tool since such changes can only result in a temporary inconsistency between the document and the tool which subsequent reprocessing by the tool must eliminate. In practice this means that each insertion point must lie within a user input region, and each fragment to be deleted or replaced must begin and end in a user input region but

```
Demo2 Front-end (version: 1.0)        File: example.8

/fol
: /*
    {\bf A sample Demo2 proof}
    to show that ``not not A'' is the same as ``A'' in 1st order logic.
    */
prove(doubleneg,(¬ (¬ #A)) ⇔ #A).

 focus: ¬ (¬ #A) ⇔ #A
 equivalence: ⇔         goal: true

/fol(window_1,theorem-top level)
: /*
    We expand the ``{\tt ⇔}'' into the conjunction of two implications
    and prove each separately.
    */
trans(def(⇔)).

 focus: (¬ (¬ #A) ⇒ #A) ∧ (#A ⇒ ¬ (¬ #A))
 equivalence: ⇔         goal: true

/fol(window_1,theorem-top level)
: /*
    We prove the 1st conjunct by transforming the ``not A'' :
    */
openwin([1,1,1]). ...

 focus: #A ⇒ ¬ (¬ #A)
 equivalence: ⇔         goal: true

openwin([1,1,1]).

 focus: ¬ #A
 2: ¬ #A
 1: #A ⇒ ¬ (¬ #A)
 equivalence: ⇔         goal: true

/fol(window_2,theorem)
: /*
    The focus is easily reducible to ``true'' because it agrees with
    hypothesis 2.
    */
trans(bool,[2]).

 focus: true
 2: ¬ #A
 1: #A ⇒ ¬ (¬ #A)
 equivalence: ⇔         goal: true
```

Figure 3: Presentation of a Demo2 proof by UQ1

may include tool generated regions within it. These edit point restrictions are trivially implemented in the augmented UQ1 editor.

To illustrate the potential for reuse enabled by these front-end editing facilities, consider the given Demo2 example. In this example, the transformations performed in the two transformation windows used are similar. Rather than retyping the commands for the second window, the user can copy the section of the development from the first window to the position where the second window will be. Modification of just two characters can then be made to the copied segment to suit the needs of the second window.

In this section, we have demonstrated how to improve the user interface of a formal method tool using a front-end editor. We were able to capture the development process and improve its presentation. Our presentation of the conceptual model of the formal method is better than that of the existing formal method tool. We have also showed how the features available in the editor can improve the review and reuse capabilities of the formal method tool.

5 Further investigations

Compared to other formal method tools, Demo2's requirements for document representation, manipulation and presentation are relatively simple. For example, the proof in each transformation window has a linear structure which is composed of a history of window states interleaved with proof commands. Windows are nested to form a simple hierarchic structure. A proof step is always carried out in a top-down style — the next window state is generated by the application of the user commands to the previous window state. This simple document structure means that hierarchic textual views can meet Demo2's current presentation requirements. A Demo2 proof can in effect be dealt with in a way similar to a program in a programming language like Pascal or Modula2. For these reasons, UQ1 can be used as the front-end for Demo2.

In general, an object involved in software development by formal methods has a more complicated structure, and requires more sophisticated manipulation and presentation. For example, a proof in Mural [11] has a graph structure in which an assertion may be related to more than one other assertion by a proof step and an assertion may be used in more than one proof step. A Mural proof can be constructed in a top-down, bottom-up or mixed style. It is obvious that textual views based on abstract syntax trees will not satisfy all the presentation requirements of a graph-structured object. For instance, the user may want to inspect the overall structure of a proof in a graphical form without the assertion details, or to navigate from the details of one assertion to another in a hypertext-like fashion.

The above requirements imply that the support provided by UQ1 does not constitute an adequate generic front-end for formal method tools. We are therefore currently investigating a more powerful generic front-end to meet formal methods' requirements for representing, manipulating and presenting software documents in their support environments. As the first step, we are developing a general user model for software documents. The following are some of its features.

1. The document structure is capable of representing hierarchic textual documents as in UQ1, and allows additional structural relations between document segments so that a graph structure can be easily accommodated.

2. A document segment can be composed of a number of other segments with a linear order or no order between them.

3. Structural constraints can be imposed on the document relations, and enforced in a number of ways, including a consistency preserving strategy, an automatic checking strategy and a when-requested checking strategy.

4. User-oriented operations on a document segment can be defined from a set of primitive operations.

5. Multiple views can be defined for a document segment according to the user's interests. They can be textual or graphical views.

Given this document model and a corresponding document description language, we propose to implement a generic document editor and demonstrate its usefulness as a front-end to a variety of formal method tools.

6 Conclusions

In this paper we have suggested that many existing tools for software development by formal methods lack user-oriented properties necessary to their effective use in software engineering. We have proposed a strategy for overcoming these deficiencies by interposing a user-oriented front-end editor between the user

and the formal method tool itself. We have demonstrated the feasibility of this approach by adapting the generic language-based editor UQ1 to provide a front-end to the theorem prover Demo2. This feasibility study shows that significant improvement in presentation, documentation, review and reuse of objects produced using tools such as Demo2 can be achieved without significant alteration to the tools themselves.

In practice, however, other tools for formal methods manipulate objects whose structure is beyond UQ1's representation capabilities. We therefore propose to define a generic model of formal method documents and to develop a corresponding front-end editor for such documents which is capable of real-time interaction with a corresponding variety of tools supporting formal methods.

Acknowledgements

We are grateful to our colleagues Warwick Allison, David Carrington, Ian Hayes and Yun Yang for their comments and suggestions, and to Peter Robinson and the interactive reasoning group at SVRC for their cooperation. This project is supported by a research grant and a post-doctoral fellowship from the Australian Research Council.

References

[1] A. Avron, F. Honsell, and I.A. Mason. An overview of the Edinburgh logical framework. In G. Birtwistle and P.A. Subrahmanyam, editors, *Current Trends in Hardware Verification and Automated Theorem Proving*, pages 323–340. Springer-Verlag, New York, 1989.

[2] R.J.R. Back. Refinement diagrams. In J.M. Morris and R.C. Shaw, editors, *Proceedings of 4th BCS-FACS UK Refinement Workshop*, Workshops in Computing, pages 125–137, Cambridge, UK, January 1991. Springer-Verlag, London, 1991.

[3] B. Broom and J. Welsh. Another approach to literate programming. In *Proceeding of 11th Australian Computer Science Conference*, pages 257–268, Brisbane, Australia, February 1988.

[4] D. Carrington and K. Robinson. A prototype program refinement editor. In *Proceedings of 3rd Australian Software Engineering Conference (ASWEC '88)*, pages 45–63, Canberra, Australia, May 1988.

[5] A.S.K. Cheng, J. Han, J. Welsh, and A. Wood. Incorporating constructive tools into a generic language-based editor. Technical Report 93-7, Software Verification Research Centre, The University of Queensland, St. Lucia, Australia, April 1993.

[6] R.L. Constable, S.F. Allen, et al. *Implementing Mathematics with the Nuprl Proof Development System.* Prentice-Hall, Englewood Cliffs, New Jersey, 1986.

[7] D. Craigen, S. Kromodimoeljo, I. Meisels, B. Pase, and M. Saaltink. EVES: An overview. In S. Prehn and H. Toetenel, editors, *VDM '91, formal software development methods, Proceedings of 4th International Symposium of VDM Europe*, volume 551 of *Lecture Notes in Computer Science*, pages 389–405, Noordwijkerhout, The Netherlands, October 1991. Springer-Verlag, Berlin, 1991.

[8] S.L. Gerhart, D.R. Musser, et al. An overview of AFFIRM: A specification and verification system. In S.H. Lavington, editor, *Information Processing 80, Proceedings of IFIP Congress 80*, volume 8 of *IFIP Congress Series*, pages 343–347. North-Holland Publishing Company, October 1980.

[9] D. Good. Mechanical proofs about computer programs. In C.A.R. Hoare and J.C. Shepherdson, editors, *Mathematical Logic and Programming Languages*, Prentice-Hall International Series in Computer Science, pages 55–75. Prentice-Hall International, London, 1985.

[10] M.J.C. Gordon. HOL: A proof generating system for higher-order logic. In G. Birtwistle and P.A. Subrahmanyam, editors, *VLSI Specification, Verification and Synthesis*, pages 73–128. Kluwer Academic Publishers, Boston, Massachusettes, 1988.

[11] C.B. Jones, K.D. Jones, P.A. Lindsay, and R. Moore. mural*: A Formal Development Support System.* Springer-Verlag, London, 1991.

[12] D.R. Musser. Abstract data type specification in the AFFIRM system. *IEEE Transactions on Software Engineering*, SE-6(1):24–32, January 1980.

[13] D.R. Musser. Automated theorem proving for analysis and synthesis of computations. In G. Birtwistle and P.A. Subrahmanyam, editors, *Current Trends in Hardware Verification*

and Automated Theorem Proving, pages 440–464. Springer-Verlag, New York, 1989.

[14] M. Nielsen, K. Havelund, K.R. Wagner, and C. George. The RAISE language, method and tools. *Formal Aspects of Computing*, 1(1):85–114, January–March 1989.

[15] P.J. Robinson and T.G. Tang. The demonstration interactive theorem prover: Demo2.1. Technical Report 91-4, Software Verification Research Centre, The University of Queensland, St. Lucia, Australia, September 1991.

[16] J. Rushby, F. von Henke, and S. Owre. An introduction to formal specification and verification using EHDM. CSL Technical Report SRI-CSL-91-02, SRI International, Menlo Park, California, February 1991.

[17] B.A. Silverberg. An overview of the SRI hierarchical development methodology. In H. Hünke, editor, *Software Engineering Environments*, pages 235–252. North-Holland Publishing Company, Amsterdam, 1981.

[18] T. Vickers. An overview of a refinement editor. In *Proceedings of 5th Australian Software Engineering Conference (ASWEC '90)*, pages 39–44, Sydney, Australia, May 1990.

[19] T. Vickers. An overview of a theorem proving assistant. In *Proceedings of 13th Australian Computer Science Conference*, pages 402–411, Melbourne, Australia, February 1990.

[20] J. Welsh, G.A. Rose, and M. Lloyd. An adaptive program editor. *Australian Computer Journal*, 18(2):67–74, May 1986.

[21] J. Welsh and Y. Yang. Tool integration techniques. In *Proceeding of 6th Australian Software Engineering Conference (ASWEC '91)*, pages 405–418, Sydney, Australia, July 1991.

A LaTeX output of a Demo2 proof

/fol
: **A sample Demo2 proof** to show that "not not A" is the same as "A" in 1st order logic.
prove (doubleneg, $(\neg(\neg A)) \Leftrightarrow A$).

focus :	$\neg(\neg A) \Leftrightarrow A$	
equivalence : $\Leftrightarrow$		goal : true

/fol(window_1,theorem-top level)
: We expand the "<=>" into the conjunction of two implications and prove each separately.
trans (def($\Leftrightarrow$)).

focus :	$(\neg(\neg A) \Rightarrow A) \wedge (A \Rightarrow \neg(\neg A))$	
equivalence : $\Leftrightarrow$		goal : true

/fol(window_1,theorem-top level)
: We prove the 1st conjunct by transforming the "not A":
openwin ([1,1,1]). $\lhd 1$

focus :	$\neg A$	
2:	$\neg A$	
1:	$A \Rightarrow \neg(\neg A)$	
equivalence : $\Leftrightarrow$		goal : true

/fol(window_2,theorem)
: The focus is easily reducible to "true" because it agrees with hypothesis 2.
trans (bool,[2]).

focus :	true	
2:	$\neg A$	
1:	$A \Rightarrow \neg(\neg A)$	
equivalence : $\Leftrightarrow$		goal : true

/fol(window_2,theorem)
: and we return the result back to the original expression.
closewin . $\rhd 1$

focus :	$A \Rightarrow \neg(\neg A)$	
equivalence : $\Leftrightarrow$		goal : true

/fol(window_1,theorem-top level)
: After substituting the transformed term, simplification reduces the 1st conjunct to true, leaving us to prove the 2nd.
This we do by a similar strategy, transforming the 2nd "A":
openwin ([2,1,1]). $\lhd 1$

focus :	A	
1:	A	
equivalence : $\Leftrightarrow$		goal : true

/fol(window_2,theorem)
: The focus is again easily reducible to "true" because it agrees with hypothesis 1.
trans (bool,[1]).

focus :	true	
1:	A	
equivalence : $\Leftrightarrow$		goal : true

/fol(window_2,theorem)
: and we return the result back to original expression.
closewin . $\rhd 1$

focus :	true	
equivalence : $\Leftrightarrow$		goal : true

/fol(window_1,theorem-top level)
: Both conjuncts are shown to be true.The proof is complete.
closeproof .