

**SOFTWARE VERIFICATION RESEARCH CENTRE
DEPARTMENT OF COMPUTER SCIENCE
THE UNIVERSITY OF QUEENSLAND**

**Queensland 4072
Australia**

TECHNICAL REPORT

No. 93-7

Incorporating Constructive Tools into a Generic Language-Based Editor

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

April 1993

Phone: +61 7 365 1003

Fax: +61 7 365 1533

Incorporating Constructive Tools into a Generic Language-Based Editor

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

e-mails: {chena, han, jim, harold}@cs.uq.oz.au

Abstract

As a uniform front-end user interface, a generic language-based editor should have the capability to incorporate both analytic and constructive back-end tools. In this paper, we analyse the requirements of, and propose a strategy for, integrating constructive tools into a generic language-based editor. The major issues addressed include definition of software documents containing tool contributions, real-time consistency checking of such documents, communication and synchronisation between the editor and constructive tools. We demonstrate the feasibility of this approach by integrating a theorem prover into the enhanced generic editor. To cater for the full range of representation and integration requirements of both analytic and constructive tools, we propose to define a generic model of software documents and to develop a corresponding front-end editor for such documents which is capable of real-time interaction with these tools.

1 Introduction

Software development tools are needed to assist the software engineer. Generic language-based editors that parse input according to a predefined language-specific syntax description, are among such tools. These editors can provide reduced input effort, immediate detection of syntax errors, and automatic formatting of software text. They can also provide a uniform interface for the user to interact with other development tools. Tools accessed through the editors can be classified as either constructive or analytic on the basis of their involvement in the construction of software documents. Appropriate output from a constructive tool forms part of the software documents while the output from an analytic tool does not.

Effective software development requires verification of the relevant semantics of software documents as well as their syntax at each stage of the software preparation process. Analytic tools, such as type checkers and semantic analysers, often provide such verification. The user of a generic language-based editor expects to be able to invoke analytic tools, either explicitly or implicitly, at any time and receive feedback from the analysis. This requires that the generic language-based editors provide support for integrating analytic tools.

The value of formal methods in the development of reliable software systems is increasingly recognised. With these methods comes the need for new tools to assist the development

process. Formal method tools, such as theorem provers [8, 9] and refinement assistants [1, 4], have been developed as stand-alone tools that run interactively with the user. Most of these tools are constructive in the sense defined above because the semantics-based output of the tools is an integral part of the software document being prepared. Because these tools actively contribute to the actual document, the integration techniques that have been developed for analytic tools do not necessarily provide appropriate support for the incorporation of constructive tools.

In a previous paper [5], we have discussed the advantages to be gained by accommodating constructive tools within the framework of a generic language-based editor, and have outlined a prototyping experiment to demonstrate the advantages achievable. In this paper, we report our work on the enhancement of the existing generic language-based editor, UQ1 [11], to support constructive tools. (As previously developed, UQ1 had the capability to integrate analytic tools but not constructive tools.) In addition, our experience in incorporating a theorem prover Demo2 [9] into the enhanced UQ1 is reported.

After presenting a summary of UQ1 and its existing support for integration of analytic tools in section 2, we identify the additional requirements for incorporating constructive tools in section 3. In section 4, we outline how UQ1 has been enhanced to support the integration of constructive tools, and in section 5 we demonstrate how this strategy has been used to incorporate Demo2 into UQ1 and discuss the limitations of the approach. We outline some future work in section 6 before concluding the paper.

2 UQ1 and Analytic Tools

The generic language-based editor UQ1 [11] 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 an editor builder. For document editing, UQ1 supports all the standard editing facilities using a direct textual manipulation paradigm. Document display is syntax-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. View formatting is performed automatically from formatting primitives embedded in the language definition, as described in [10]. UQ1 is also capable of generating a formatted \LaTeX document from the input.

2.1 Analytic Tool Integration

Apart from the syntax-based manipulation of the document, UQ1 allows the user to invoke analytic, typically semantic, tools. To do so, UQ1 needs to coordinate the information flow among the user, the display, and the tool. Figure 1 shows the conceptual information flow among these components. The user input is parsed according to an input grammar to create the stored internal representation of the document S . The on-screen display is generated by applying a transformation u_1 , usually called an unparsing schema, to S . The representation processed by the tool is generated by applying the transformation u_2 to S .

The tools typically generate responses to indicate the result of the analysis. These responses do not form a part of the document being prepared. Typically they consist of messages which

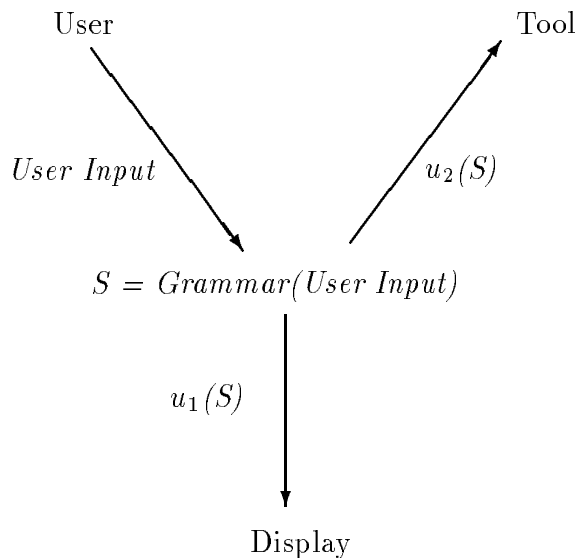


Figure 1: Information Flow between the Front-end and an Analytic Tool

relate to specific parts of it. Therefore, an effective mechanism for mapping the messages to the displayed document representation is all that is required.

To enable flexibility in both the construction of new analytic tools and the reuse of existing tools, UQ1's design maintains a physical separation between the generic editor (called the front-end) and the integrated analytic tools (called back-end tools), as shown in Figure 2.

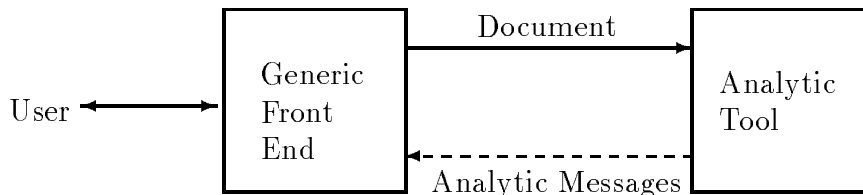


Figure 2: Separation of the Front-end and an Analytic Tool

The exact nature of this separation depends on the integration paradigm in use. There are three paradigms available in UQ1 [12]. The simplest method is the uncoupled paradigm, in which the front-end editor and the back-end tool are separately executable programs which communicate via files. The front-end generates an intermediate ASCII file by unparsing the internal document representation. In this case, therefore, transformation u_2 in Figure 1 is also an unparsing schema. This file is then parsed by the tool to construct the representation required for analysis. Feedback filed by the tool is integrated in the front-end display by a reverse mapping from the line and character positions in the ASCII file to the front-end's representation.

The directly contrasting approach to the uncoupled paradigm is the tightly-coupled paradigm. The front-end and the tool are combined as a single executable program by linking separately compiled modules. The internal document representation and the display are effectively

shared by both the front-end and the tool. In this case, the transformation u_2 enables direct tool access to the syntax tree maintained by the front-end editor, as an abstract data type. Mapping tool feedback onto the front-end’s representation is trivial in this case.

The loosely-coupled paradigm is a compromise between the last two approaches. The front-end and the tool are separate, concurrently executing processes which communicate via messages over an interprocess communication channel. In this case, transformation u_2 determines the encoding of updates, as transmitted by the front-end, to an abstract syntax tree maintained by the loosely-coupled tool. Feedback messages from the tool are decoded with respect to the front-end’s displayable representation by the same mechanism.

2.2 Document Consistency Checking

UQ1 supports a mixed paradigm in checking the consistency of the document under preparation. During user input, syntax errors in newly input material are detected and rejected immediately by the parser. In effect, the user cannot create syntax errors during input. This constraint automatically applies to new material inserted in an existing document, but editing operations can create syntactic inconsistencies downstream from the point of edit. Allowing such intermediate inconsistencies to exist is necessary with a text-editing paradigm. By default, UQ1 adopts a *lazy* strategy to the detection of downstream inconsistencies — they are detected and brought to the user’s attention only when some edit operation further down the document depends on their elimination. This concept is reflected in Figure 3 which shows the document is guaranteed to be syntactically correct up to such an editing position. In addition, the user can request a global syntax check at any time to determine whether any such inconsistencies exist.



Figure 3: Syntactic Consistency of a Document

Semantic checking by analytic tools is carried out only when the user requests. This is seen as the preferred user option in most circumstances — users are more often irritated than assisted by the feedback produced by automatic incremental semantic checking. We note, however, that the tightly- and loosely-coupled paradigms as implemented are readily adaptable to automatic incremental semantic checking. Changes to the document are already signaled incrementally to the tool to allow incremental analysis to take place. Only the feedback strategy need to be changed to enable automatic incremental checking from the user’s viewpoint. By its nature, file-based uncoupled integration cannot effectively support automatic incremental checking. With the current implementation, whenever a semantic check is requested, an automatic syntactic check for the entire document is first performed. Only when the document is syntactically consistent is the analytic tool invoked.

3 Requirements for Incorporating Constructive Tools

The purpose of integrating a constructive tool with a generic language-based editor is to combine the editor's syntax-based editing and display capabilities with the constructive semantic capabilities of the tool. The strategy may be advantageous in enhancing the editing and display capabilities of an existing constructive tool, in reducing the development cost of a new constructive tool, or simply in achieving a higher degree of integration between a set of constructive and analytic tools.

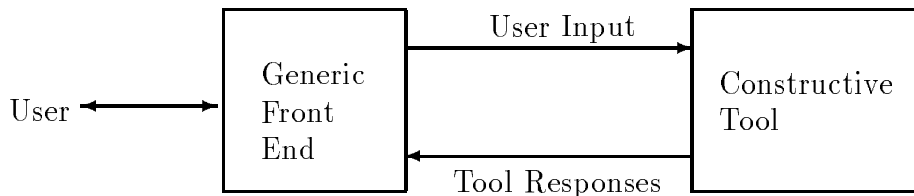


Figure 4: Relationship between the Generic Front-end and a Constructive Tool

Figure 4 shows the architecture for the integration of a constructive tool. The front-end provides user-oriented support for input, editing and display of the document under construction. User input relevant to the tool is transmitted by the front-end to the tool. Similarly, the responses from the tool are relayed back to the front-end, but these responses now contribute directly to the document, and determine the user's next input options. As both the user inputs and the tool responses form part of the document, the front-end editor must be able to parse and display both.

In developing the document, the user and the constructive tool effectively conduct a dialogue via the front-end editor. The user inputs a constructive step which is relayed to the tool; the tool responds; the user inputs the next step; the tool replies; and so on. This dialogue typically proceeds at a relatively fine-grained level, and user satisfaction depends on the observed responsiveness of the tool. The communication mechanism between the front-end editor and the tool must therefore be appropriate to such fine-grained, time-sensitive dialogue.

The user and the constructive tool work cooperatively in constructing a software document. This cooperation is recorded in the document by the front-end. The essential benefit of this approach is that the user can subsequently review and reuse the recorded information in whole or in part. Editing is an integral part of the reuse process. Using the front-end's facilities, the user can move freely within the document to perform any editing appropriate to current needs. However, two additional requirements arise from such unconstrained editing.

Firstly, in an insertion operation the user expects the same tool responses as would arise during initial document input. Synchronisation is therefore needed to reconcile the *point of attention* (or state) of the tool with that of the editor as the insertion gets under way.

Secondly, any edit operation may induce downstream inconsistencies in the semantic content of the tool responses, as well as in the syntactic structure of the document. An augmented consistency checking strategy is therefore necessary, to deal with both syntactic and semantic consistency after changes are made. Such consistency checking will in general require reprocessing of the document by the tool as well as reparsing by the front-end editor.

In practice, it is sensible to impose some constraints on user editing of an existing document.

In most cases, it is illogical to allow the user to edit information generated by the tool. Such changes can only result in an inconsistency between the document and the tool, which subsequent reprocessing by the tool must eliminate.

4 Incorporating Constructive Tools into UQ1

In extending UQ1 to support integration of constructive tools, it is logical to reuse both the existing conceptual model of tool integration and the existing implementations of the various integration paradigms as far as possible.

Figure 5 shows the required information flow among the user, the display, and a constructive tool. The user input is parsed according to an input grammar $Grammar_u$ to create the user-defined component S_u of the stored internal representation of the document. This representation of the user input is translated into a form required by the constructive tool by applying the transformation u_2 to S_u . The resulting tool response is parsed according to a tool output grammar $Grammar_t$ to create the tool-generated component S_t of the stored internal representation of the document. The on-screen display is an interleaving of the user inputs and tool responses generated by applying the unparsing schema u_1 to S_u and the unparsing schema u_3 to S_t .

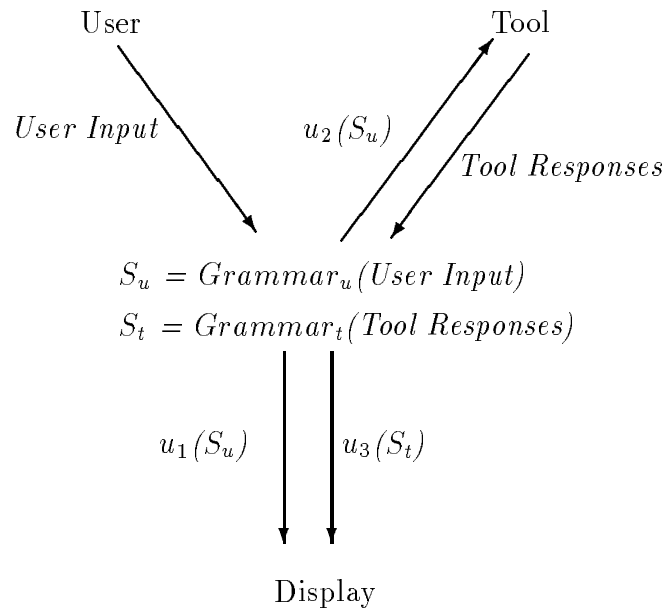


Figure 5: Information Flow between the Front-End and a Constructive Tool

In practice, the extended UQ1 must realise this extended conceptual information flow by adapting and extending its existing language definition facilities, its parsing mechanisms and its tool integration mechanisms to meet the requirements set out in the preceding section.

4.1 Defining the Syntax

Conceptually, the grammar for the user input S_u and the grammar for the tool responses S_t define separate input streams to be processed by the front-end editor, but in practice, it is better to integrate them to reflect the way in which these streams are interleaved in the resulting document. Within this integrated syntax, however, the parser must be able to distinguish segments input by the user from those input by the tool.

The grammar of the language to be processed by UQ1 is specified in a grammar definition file. This file is first input to a *Rule Filer* which validates the grammar and stores it in a conveniently loadable form. The UQ1 editor then uses this intermediate file to create its internal representation of the language grammar.

The grammar in the grammar definition file is expressed in an augmented Extended Backus-Naur Form (EBNF). The EBNF has been enriched by formatting directives, help rules, non-terminal attributes, lookahead paths, and restart markers. Each syntax rule may have zero or more *attributes* associated with it. Attributes are used to indicate that a particular rule has special properties (such as for *lookahead* and *help* rules) or to denote special display, help, or tool related information for the associated non-terminal.

To differentiate within the language definition between the input from the user and the input from the constructive tool, the obvious solution is to tag the productions in the grammar that represent contributions from the tool with a *tool* attribute, **-T**. For example, a simple dialogue document might be defined as follows:

```
Dialogue      = Tool { User Tool } .
Tool          -T = ToolInfo Prompt .
User          = ...
```

By examining the attributes of non-terminals expanded, the parser is then able to determine whether to take subsequent input symbols from the user or from the constructive tool.

In fact, the parser must decide when a change of input source is needed without requiring further symbols from the current source. It is crucial, therefore, that there is no situation where the source of the next symbol is ambiguous. In parsing terms, the language must be LL(0) at the *change-over* positions since no lookahead is allowable. From a language design viewpoint, each user or tool contribution must be “self-terminating” to enable this grammar constraint to be met. The rule filer should check that the grammar filed meets this condition. Currently the editor builder must be clearly aware of this constraint in designing the language concerned and preparing the grammar that defines it.

4.2 Implementing Communication

Implementing the communication between the UQ1 editor and a constructive tool involves a number of design decisions relating to the actual communication mechanism used, the strategy for dispatching user input to the tool, and the strategy for dealing with tool responses.

4.2.1 The Communication Paradigm

The three existing integration paradigms for analytic tools use three distinct data communication mechanisms: file writing and reading, shared data structures, and interprocess

communication channels. Information is communicated in two forms: ASCII document text and abstract syntax trees.

Each of these combinations has its particular advantages and disadvantages with respect to the range of existing tools that can be integrated, the ease with which the integration is achieved, and the performance of the resultant combination [13]. In principle, the same advantages and disadvantages are equally relevant to constructive tools, and provision of suitably adapted versions of each paradigm is a logical strategy to consider.

The use of ASCII text, as in the uncoupled paradigm for analytic tools, offers the most flexibility for reuse of existing tools. However, the file writing and reading used in the uncoupled paradigm is incapable of supporting the fine-grained dialogue required by constructive tools. Direct interprocess communication, as used in the loosely-coupled paradigm, can easily support such fine-grained dialogue. As a first version of a constructive tool integration paradigm, therefore, the combination of sending ASCII text via interprocess communication channels seemed the obvious choice. Investigation of alternatives based on abstract syntax tree exchange was postponed pending successful demonstration of the text-based paradigm.

4.2.2 Editor—Tool Communication

The communication between the editor and the tool is bi-directional. For the communication from the editor to the tool, we need to determine what information to send, and when to send it.

In principle, all the user input since the previous transmission might be sent to the tool. However, the user input may contain information which is irrelevant to the tool, such as user documentation which is included only to improve the readability of a given document. Such information should be hidden from the tool. To identify such irrelevant information, non-terminals representing it in the EBNF are tagged with the *hidden* attribute, **-H**.

User	=	Commentary Command.
Commentary	-H =	...
Command	=	...

This ability to hide irrelevant information is a general enhancement to the UQ1 editor that can be used in both constructive and analytic tools. (For constructive tools, however, the hidden attribute can also be used to hide irrelevant information generated by the tool, so preventing its capture and display in the document.)

The user information can be sent to the tool either *eagerly* or *lazily*. With the eager approach, each fragment of user input is sent to the tool as soon as possible. The lazy approach, on the other hand, does not transmit the user input to the tool until absolutely necessary, i.e., when the user-to-tool change-over position is reached. In either case, the user input may have to be transformed into the format expected by the tool before transmission.

The eager approach has the advantage of being simple to implement, and of optimising the observable tool response time. The lazy approach, however, has the important consequence that the current segment of user input can also be edited without affecting the synchronisation of the tool. The lazy approach has therefore been implemented in UQ1.

To implement the physical communication involved, a Unix pseudo-terminal rather than a simple pipe is chosen as the communication channel. A pseudo-terminal provides the same

I/O characteristics as a terminal. This allows an existing constructive tool to operate in the same fashion as before the integration. Likewise, the editor can handle tool contributions in the same manner as it handles user input.

4.2.3 Abnormal Tool Responses

All the information that is sent by the editor to the tool is guaranteed to be syntactically correct but is not necessarily semantically correct. When semantically incorrect information is sent to the tool, the tool will normally respond with an error message. In general, the semantics of inputs to date, as perceived by the tool, may determine the allowable continuations, both semantic and syntactic, for the document concerned. In the presence of a semantic error, the allowable syntactic continuation may be undetermined. To maintain UQ1's existing paradigm with respect to syntax errors during input implies that such semantic errors should be rejected for user correction before input can continue. Upon the receipt of an abnormal tool response, therefore, the editor should interrupt the current input sequence, highlight the user input that caused the error, and display the relevant error message. The user must then determine the action to be taken to resolve the error.

The conventions used by the constructive tool to distinguish its normal and abnormal responses may vary widely. To avoid unduly constraining assumptions by the generic front-end, the chosen approach is to define the abnormal responses as productions in the document grammar, with some default syntactic continuation. To distinguish these from other productions, they are tagged with the *error* attribute, **-E**.

```
ToolInfo          = NormalInfo | AbnormalInfo.  
NormalInfo       = ...  
AbnormalInfo     -E = ...
```

Whenever these error productions are applied in parsing the tool response, the error handling strategy described above is automatically invoked. Appropriate corrective action by the user will lead to automatic elimination of the error productions concerned, but any attempt to make downstream changes, including resumption of input immediately beyond the error, will be blocked by the presence of these productions in the document.

4.3 Editing and Synchronisation

As document input proceeds, the constructive tool's point of attention is the last change-over point from tool input to user input. An attempted edit operation may result in a difference between the points of attention in the editor and the tool. Synchronisation is needed to reconcile this difference. The synchronisation strategy used depends on the position of the tool relative to the editing point.

If the tool is upstream relative to the editing point, all user inputs between the tool position and the editing point are sent to the tool. As the tool processes the user inputs, the state of the tool is advanced. Eventually the tool will reach the same position as the edit.

If the tool is downstream relative to the editing point, it is necessary to reset the tool to some position before the editing point, and replay the user inputs between that position and the editing point.

To enable incremental reparsing of documents, the current implementation of UQ1 has provision for the inclusion of restart markers in the EBNF. These restart markers indicate positions for recoverable states for the parser. At each document position corresponding to a restart marker, the parser saves a copy of its current state as an available restart point. When an edit operation is attempted, the parser is wound back to the closest available restart point before the edit operation, and the syntactic check is started from there. Restart points which are downstream of an edit operation are invalidated by the operation and become unavailable.

The same mechanism can be used to synchronise a constructive tool. Restart markers are now associated with the immediately preceding *tool-to-user* change-over positions. When the parser is reset to a given restart point, the constructive tool must be manipulated to ensure that the document up to the associated *tool-to-user* change-over position is semantically consistent and to enter the corresponding state. How this is achieved depends on the incremental capabilities of the tool — in the worst case it may have to reprocess the document from the start to the change-over position required.

4.4 Consistency Checking

With analytic tools in a UQ1 environment, the continuation of document development depends only on the syntactic consistency of the document. Each edit operation automatically ensures the necessary degree of syntactic consistency exists but, alternatively, the user may request a syntactic consistency check at any time. Semantic consistency is a distinct concept whose checking is only initiated by the user.

For constructive tools, however, a necessary relationship between syntactic and semantic consistency is maintained, as implied by the discussion of synchronisation in section 4.3. The syntactic and semantic checks for the document commence together and proceed forward through the document. At any given time, some section of the document is syntactically correct and some section is semantically correct. But, the semantic check can never proceed past the syntactic check. This is illustrated in Figure 6. Each development step depends

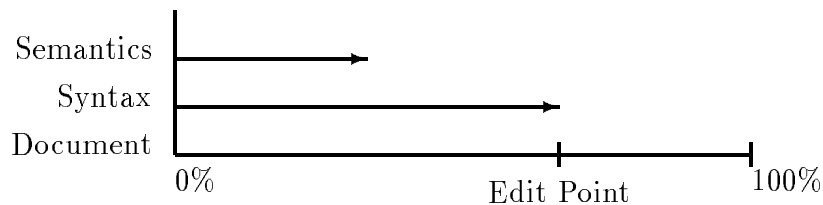


Figure 6: Syntactic and Semantic Consistency of a Document

on both the syntactic and semantic consistency of inputs to date, and each edit operation automatically ensures that the necessary degree of syntactic and semantic consistency prevails. In these circumstances it is illogical to offer the user syntactic and semantic consistency checking as separate functions — a single consistency check is appropriate.

When inconsistencies are detected, either during a user-initiated check or as a result of an attempted edit operation, both syntactic and semantic errors are handled by the protocol described for abnormal tool responses in section 4.2.3. From the user's viewpoint, however, a third kind of inconsistency can arise.

Consistency checking in general involves the reprocessing of some or all of the document by the constructive tool. As each existing user contribution is reprocessed by the tool, the tool response may differ from the tool response previously recorded in the document. In some cases the user may have anticipated this difference in which case the new response should simply replace the old in the document. In other cases the user may not have anticipated the difference and may wish to take action as a result of it, or at least to consider why the difference has arisen. In principle, therefore, the editor might handle this “inconsistency” by one of the following actions:

1. Accept the new tool response without consulting the user.
2. Terminate the consistency checking and report the discrepancy.
3. Inform the user about the discrepancy and enquire how to handle this and subsequent discrepancies.

From the user’s viewpoint, the first option is preferable in situations where, for example, the name of a variable has been changed systematically. It avoids the frustration caused by having the user acknowledge every corresponding correction in the tool responses. The second option is preferable in situations where the first discrepancy is indicative of further unintended discrepancies in the rest of the document. The last suggestion provides more flexible user control of handling discrepancies, but may still be unsatisfactory since different types of discrepancy may need different handling. Providing user-definable modes of discrepancy handling may be the most effective design choice. For the moment, the first option is used in UQ1 because it is simple to implement.

5 A Case Study

In this section, we present a case study which integrates the interactive theorem prover Demo2 as a constructive tool for UQ1. We also discuss the limitations of the integration strategy presented in the previous section with respect to the case study.

Demo2 [9] 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, and 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. An example Demo2 proof is shown in Figure 7.

In the integrated system produced by the case study, Demo2 proofs are captured as UQ1 documents [5]. These documents include the normal user inputs to Demo2, the Demo2 responses, and additional user documentation. In this way, the basic proof construction process via Demo2 remains essentially unchanged (as UQ1 document input), but the resultant proof is now documentable, reviewable and reusable via the browsing and editing facilities of UQ1.

Demo2 Session	Commentary
<pre> /fo1 : prove(doubleneg, (not (not #A)) <=> #A). focus: not (not #A) <=> #A Equivalence: <=> Goal: true /fo1 (window_1,theorem - top level) : trans(def(<=>)). focus: (not (not #A) => #A) and (#A => not (not #A)) Equivalence: <=> Goal: true /fo1 (window_1,theorem - top level) : openwin([1,1,1]). focus: not #A 2: not #A 1: #A => not (not #A) Equivalence: <=> Goal: true /fo1 (window_2,theorem) : trans(bool, [2]). focus: true 2: not #A 1: #A => not (not #A) Equivalence: <=> Goal: true /fo1 (window_2,theorem) : closewin. focus: #A => not (not #A) Equivalence: <=> Goal: true /fo1 (window_1,theorem - top level) : openwin([2,1,1]). focus: #A 1: #A Equivalence: <=> Goal: true /fo1 (window_2,theorem) : trans(bool, [1]). focus: true 1: #A Equivalence: <=> Goal: true /fo1 (window_2,theorem) : closewin. focus: true Equivalence: <=> Goal: true /fo1 (window_1,theorem - top level) : closeproof. </pre>	<p>Begin the proof on the expression.</p> <p>Demo2's window state showing current expression, equivalence being employed, and goal to achieve.</p> <p>Expand the definition of “<=>” using “=>” and “and”.</p> <p>Concentrate on the 1st conjunct. Open a window on the 1st “not #A”.</p> <p>Hypotheses deduced by Demo2 from the previous focus. Note that hypothesis 2 is an immediate match for the focus.</p> <p>Transform using boolean introduction.</p> <p>Focus and the hypothesis agree and result in “true”.</p> <p>Return the result.</p> <p>Substitute the result and simplify the expression — the first conjunct is proven.</p> <p>Now concentrate on the 2nd conjunct. Open a window on the 2nd #A.</p> <p>The deduced hypothesis again offers an immediate solution.</p> <p>Transform using boolean introduction.</p> <p>Focus and hypothesis agree and result in “true”.</p> <p>Return the result.</p> <p>The expression is proven to be true.</p> <p>Complete the proof.</p>

Figure 7: A Typical Demo2 Proof Session

In achieving this integration, the augmented UQ1 facilities described in section 4 have proved simple to use.

Figure 8 is a screen dump of the UQ1 document for the Demo2 proof shown in Figure 7. It shows how

- UQ1’s nested context structure has been used to bring out the window nesting implicit in Demo2 proofs — the upper view shows the top-level transformation sequence in the proof, while the lower view shows the transformation sequence within the inference window opened in the upper view;
- additional user documentation has been incorporated in the UQ1 document, but remains hidden from Demo2;
- presentation of Demo2 symbols and constructs has been improved using UQ1’s symbol transformation and box-drawing capabilities.

Appendix A shows the corresponding hard copy of the Demo2 proof shown in Figure 7, produced using the UQ1 facility for generating typesettable L^AT_EX documents. In this version, the printed order is that of the original Demo2 dialogue, i.e., with the window inferencing subsequences embedded in the main transformation sequence, with margin indicators to show the nesting involved.

An alternative form of printout is that illustrated in [2, 3] where the window inferencing sequences are printed after the sequence in which they are nested, with automatic cross-referencing between their point of reference and their actual printout.

This case study has demonstrated how Demo2 can be integrated with the enhanced UQ1 editor, to achieve significant improvement in the presentation, review and reuse of Demo2 proofs without alteration to Demo2 itself. However, not all problems arising from Demo2 have successfully been dealt with using the UQ1 facilities described.

In integrating Demo2 with UQ1, we have retained the existing rather clumsy Demo2 notation for referring to subterms of the term under transformation or to hypotheses in the current hypothesis list. In opening the first inferencing window, for example, [1,1,1] refers to **not #A** as the first subterm of the first subterm of the first subterm of the current focus! The natural way to input this reference is for the user to click on the relevant subterm while inputting the **openwin** command. In UQ1 document terms, however, the user is creating a reference or *relation* between the **openwin** command and the subterm concerned, and this relation must remain visible to subsequent review or reprocessing of the document. At present UQ1 has no capacity to record and display such intra-document relations. For proof systems other than Demo2, and indeed for the capture and presentation of a wide range of document classes, this limitation may be much more significant. It is one which we propose to address in future work.

Like most theorem provers [7, 8], Demo2 supports macros and tactics. These facilities enable the tool users to develop sophisticated proof methods from the basic commands. Our strategy has not successfully addressed the issue of presenting tactic applications within the existing proof structure, especially when the tactics start and terminate at different nesting levels. In effect, a structure clash arises between the user’s potential conceptual models of the proof involved — on the one hand the hierarchy of nested inference windows is a powerful model for the overall structure of the proof process, but on the other hand the concept of a tactic application as a high-level encapsulated proof operation cuts across this nested window model.

```

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,( $\neg (\neg \#A)$ )  $\Leftrightarrow$   $\#A$ ).
├── focus:  $\neg (\neg \#A) \Leftrightarrow \#A$ 
├── equivalence:  $\Leftrightarrow$       goal: true
└─ /fol(window_1,theorem-top level)
  : /*
  We expand the ``{\tt  $\Leftrightarrow$ }`` 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]). ...
├── focus:  $\#A \Rightarrow \neg (\neg \#A)$ 
├── equivalence:  $\Leftrightarrow$       goal: true
└─ openwin([1,1,1]).
  ├── 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

```

Figure 8: Presentation of a Demo2 Proof by UQ1

Successful resolution of such clashes is ultimately an issue in conceptual system design, but becomes more significant when one attempts to capture the overall model of a development process in a corresponding explicit document structure. Further study of this problem is vital to the use of front-end editors such as UQ1 to capture dialogues with constructive tools such as Demo2 as reviewable, reusable documents.

6 Future Work

As discussed in the previous section, the proposed integration strategy for constructive tools has limitations in meeting the requirements of providing user-oriented support for the Demo2 theorem prover via the generic UQ1 front-end. Furthermore, Demo2's requirements for document representation and tool integration are relatively simple, compared to other constructive tools. For example, the proof in a Demo2 transformation window has a linear structure of interleaved window states and proof commands, and is constructed sequentially in a top-down style. Whereas, a proof in Mural has a graph structure with assertions as nodes and proof steps as edges, and can be constructed in a top-down, bottom-up or mixed style. It is also apparent that a graph-structured object is more difficult to present than a sequence-structured object.

With the experience of the work reported here and after a systematic analysis of constructive tools, especially formal method tools [6], we are currently investigating a more powerful generic front-end to meet their requirements for document representation and tool integration. As the first step, we are developing a general user model for software documents developed with the assistance of analytic and constructive tools. 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. Structural constraints can be imposed on the document relations, and enforced according to a consistency preserving strategy, an automatic checking strategy or a when-required checking strategy.
3. The document manipulation mechanism allows integration of multiple analytic and constructive tools. User-oriented semantic operations on a document can be defined using functions provided by these tools, in a flexible manner.
4. 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 analytic and constructive tools.

7 Conclusion

In this paper we have suggested that as a uniform front-end user interface, a generic language-based editor should have the capability to integrate both analytic and constructive back-

end tools. Based on the existing techniques for integrating analytic tools into the generic language-based editor UQ1, we have analysed the additional requirements of, proposed and implemented a strategy for, integrating constructive tools into UQ1. The major issues addressed include definition of software documents containing tool contributions, real-time consistency checking of such documents, communication and synchronisation between the editor and constructive tools. We have demonstrated the feasibility of this approach by integrating the theorem prover Demo2 into the enhanced UQ1. This feasibility study shows that the proposed approach can meet the major requirements for integrating constructive tools such as Demo2 without significant alteration to the tools themselves.

In practice, however, other constructive tools involve software objects whose structure and manipulation are beyond the enhanced UQ1's capabilities for document representation and tool integration. We therefore propose to define a generic model of software documents and to develop a corresponding front-end editor for such documents which is capable of real-time interaction with a corresponding variety of analytic and constructive tools.

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] 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.
- [2] B. Broom and J. Welsh. Another Approach to Literate Programming. In *Proceedings of 11th Australian Computer Science Conference*, pages 257-268, Brisbane, Australia, February 1988.
- [3] B. Broom, J. Welsh and L. Wildman. A Literate Rigorous Program Case Study. In *Proceedings of 5th Australian Software Engineering Conference (ASWEC '90)*, pages 321-326, Sydney, Australia, May 1990.
- [4] D. Carrington and K. Robinson. A Prototype Program Refinement Editor. In *Proceedings of 3rd Australian Software Engineering Conference (ASWEC '88)*, Canberra, Australia, May 1988.
- [5] A.S.K. Cheng, J. Han, J. Welsh and A. Wood. Providing User-Oriented Support for Software Development by Formal Methods. To appear in *Proceedings of 6th International Workshop on Computer-Aided Software Engineering*, Singapore, July 1993.
- [6] J. Han. *A Structural Model for Methodology-Based Interactive Rigorous Software Development*. Department of Computer Science, The University of Queensland, St. Lucia, Australia, March 1992.

- [7] 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, Massachusetts, 1988.
- [8] C.B. Jones, K.D. Jones, P.A. Lindsay, and R. Moore. *Mural: A Formal Development Support System*. Springer-Verlag, London, 1991.
- [9] 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.
- [10] G.A. Rose and J. Welsh. Formatted Programming Languages. *Software Practice and Experience*, Vol 11, pp651-669, 1981.
- [11] J. Welsh, G.A. Rose, and M. Lloyd. An Adaptive Program Editor. *Australian Computer Journal*, 18(2):67-74, May 1986.
- [12] 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.
- [13] Y. Yang, J. Welsh and W. Allison. Supporting Multiple Tool Integration Paradigms within a Single Environment. To appear in *Proceedings of 6th International Workshop on Computer-Aided Software Engineering*, Singapore, July 1993.

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, $(\neg (\neg A)) \Leftrightarrow A$).

<1Proof

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]).

<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 .

▷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]).

◁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 .

▷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 .

▷Proof