SOFTWARE VERIFICATION RESEARCH CENTRE

DEPARTMENT OF COMPUTER SCIENCE

THE UNIVERSITY OF QUEENSLAND

Queensland 4072
Australia

TECHNICAL REPORT

No. 94-2

Integration of semantic tools into document editors

Jim Welsh and Yun Yang

1994

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

Integration of semantic tools into document editors

Jim Welsh and Yun Yang

Software Verification Research Centre
The University of Queensland, Australia 4072

Abstract

Integration of the tools used to prepare and verify software documents is vital to the effectiveness of the software development process. This paper identifies the key requirements of such integration, outlines three possible paradigms for integration of semantic verification tools with document editors, and reports on their prototype implementation and evaluation with respect to these requirements.

1 Introduction

Software development is a process which involves the *preparation* and *verification* of the software documents which capture the software product concerned. An effective environment for software development is one which supports the preparation and verification of software documents by appropriate *tools*. The nature of these tools, and the way in which they are *integrated* to enable their effective use, is the focus of this paper.

In section 2, we review the preparation and verification processes in more detail, to determine basic integration requirements for the tools concerned from their user's viewpoint. In section 3, we note briefly that the environment builder has other additional concerns. In section 4, we propose an initial integration architecture in which to explore alternative integration options. In section 5, we identify three potential integration paradigms to be used within this architecture, and outline their implementation in an experimental environment. Section 6 describes some integration experiments carried out using these implementations, while section 7 summarises performance measurements carried out on some of the resultant environments. Section 8 then presents an overall qualitative evaluation based on this experience. Finally, in section 9, we discuss modifications to the architecture used, in the light of additional requirements that may apply.

2 Preparing and verifying software documents

By preparation we mean any activity which creates or modifies a document. Such activities may include initial input and subsequent modification of the document by a developer, or automatic generation of (parts of) the document by a computer-based tool. Input and modification also involves the use of computer-based tools by the developer, such as editors and related tools for perusing and understanding the documents concerned[1].

By verification we mean any activity which establishes a document's semantic consistency either with the methodological rules and constraints that apply or with the requirements

[1]Since this developer is the user of the tools concerned, we use the word *user* hereafter.

of the software's intended application[2]. Such activities may include inspection of the documents themselves, reasoning (either formally or informally) about their contents, analysing their content with the aid of computer-based tools, and testing executable programs derived from them.

Verification of software documents is therefore achieved either by human inspection (with the help of appropriate perusal support) or by subjecting the document to verification tools such as static checkers, compilers, interpreters or verification condition generators. These tools may be seen as providing the user with *feedback* on the documents concerned, to guide their further input or modification.

Feedback-generating tools may be applied in so-called *batch mode*, i.e., when a complete set of changes to a document have been carried out, or *intermittently* in the sense that the tool is applied at intervals during the document editing, either at the user's request or automatically as each change takes place.

Whether the tool carries out its verification *incrementally*, i.e., by processing only the changed parts of the document at each step, is a separate performance issue which may also be of concern. In general, ensuring that verification tools process no more of the documents than necessary after any change ensures process efficiency. The converse requirement, to ensure that all documents or parts of documents that need to be processed do get processed after any change, is of equal or greater concern.

In all cases, effective support for tool-based verification requires that the user has appropriate control over the tools concerned (which may imply either explicit or automatic activation of them), that the documents concerned are readily available to the tools in an appropriate form, and that the feedback produced by the tools is delivered to the user in an appropriate manner. These may be seen as the *control integration*, *data integration* and *presentation* or *user-interface integration* requirements identified by Wasserman [19] and by Clément [5] elsewhere. Together, these define the requirements for integration of preparation and verification tools. Within control and data integration, however, we note the specific need to support incremental processing of changes by verification tools, to meet the user's reasonable performance expectations during document manipulation.

In summary, therefore, we identify two generic concerns for verification tool support, from the user's viewpoint:

(a) *feedback behaviour*
An appropriate degree of control and especially feedback integration should exist between preparation and verification tools.

(b) *response time*
The control and data integration mechanisms should ensure that all unnecessary overheads are avoided, and that incremental response by verification tools is achievable, i.e., that no more than the necessary (re-)processing of software documents need be carried out after any change.

3 The environment builder's viewpoint

In principle, the user is indifferent to the implementation strategy used to achieve these benefits, i.e., whether they are achieved by "integration" of available components or by

[2] In this sense, our use of *verification* encompasses both verification and validation as commonly used in software engineering.

(possibly monolithic) re-implementation of the entire environment. In practice, of course the user must also be aware of the cost of obtaining the benefits, and in particular, the potentially prohibitive cost that re-implementation implies. The cost/benefit equation is therefore of some significance to the user, but is considered here primarily as the environment builder's concern.

From the environment builder's viewpoint, the expected benefits of the integrated tool set strategy, compared to a monolithic implementation, say, are primarily cost-related: the net saving is implementation cost arising from each existing tool that can be reused, and the flexibility in environment evolution which the integration strategy provides. Both these benefits impinge on cost – the first on the cost of initial environment construction or subsequent addition of new facilities, the second on the cost of adapting existing environment facilities to changing user needs.

A particular kind of flexibility which the environment builder requires is the flexibility to integrate (within a single environment) tools whose individual implementation technologies are directly incompatible. For example, a document editor may be written in C or C^{++}, a static checker may be written in a functional language like ML while a reasoning tool may be implemented in a logic programming language like Prolog. Direct combination of the implementation code of such tools is impractical in many compilation/linkage schemes, but effective cooperation between such tools remains a realistic expectation for tool integration mechanisms. Integration of existing tools which are only available in binary executable form is a commonly occurring but extreme technological constraint, where integration must be achieved without any change to the component tool concerned, or any linkage of its code to other parts of the environment.

In summary, therefore, we identify two generic concerns for verification tool support from the environment builder's viewpoint:

(c) *integration cost*
The effort required to integrate each tool should be minimised.

(d) *independence between components*
Constraints on the implementation technology used for component tools should be avoided as far as possible.

4 A conceptual tool integration architecture

As a basis for investigating different paradigms of tool integration, we use the conceptual architecture [26, 24, 23] shown in Figure 1.

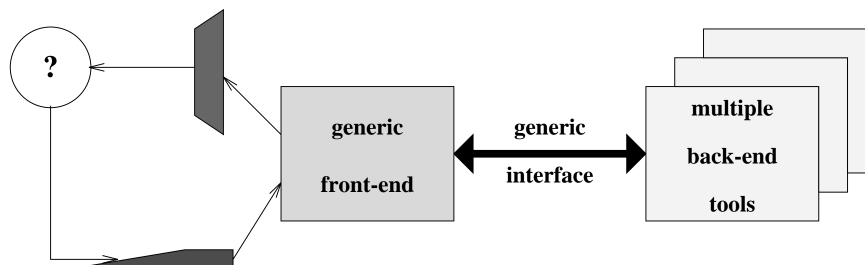

Figure 1: The conceptual architecture

The rationale for this architecture is as follows. Capabilities such as document input and editing, navigation, parameter definition, and software display, are all candidates for inclusion in the front-end, giving the user a uniform interface for a range of software-related activities. Use of a single general-purpose front-end in this way effectively meets the basic needs of *user interface integration* in that the user has a single framework for activation of the service tools involved, and feedback from these tools can in principle be collated and displayed by the front-end. The diversity of service support needed is met by the range of back-end service tools that can be activated via the front-end. The means (and ease) of adding such service tools is determined by the **generic interface** provided, which is in effect responsible for *control* and *data integration*.
Such an architecture may be seen as an extension of the currently advocated design strategy for interactive systems, which separates user interface and functional application concerns into presentation and application layers. Our front-end is a user interface with built-in knowledge of the editing and display primitives appropriate to software objects, which separates these concerns from the service applications that deal with the meaning or semantics of the objects under manipulation.
Our assumption that a single general-purpose front-end is sufficient to meet the user's editing and display needs is in direct contrast to multiple concurrent view environments such as MultiView [8] and PECAN [13]. The architecture of these environments is specifically designed to allow simultaneous manipulation of a software object via multiple view processes, each with its own edit and display paradigm. The generic front-ends developed at the University of Queensland to date, UQ1 [22] and UQ2 [3], support the projection of multiple views of software objects, but with a common edit and display paradigm. Projection of (read-only) views based on other display paradigms can be incorporated via additional back-end tools. We feel, however, that the desirability of edit capabilities on such views remains an open question. At this stage, therefore, we retain an architecture in which all user editing of software objects occurs through a single front-end. In section 9, we discuss a variation on this architecture which would admit multiple front-ends if required.
The purpose of the architecture described is to offer the user a uniform interface to a range of software development activities. In principle, the *front-end* tool needs to be no more than a text editor with the capacity to control and feed the *back-end* verification tools involved. Environments based on the *emacs* editor [17] are examples of this approach. We note, however, that the inclusion of language knowledge in the front-end of the architecture proposed above is of immediate relevance to how communication is achieved between the front-end and the back-end tools, which is the primary topic of this paper. We therefore assume a *language-based* editing facility for the front-end as shown in Figure 1 and note that the nature of this language-based editing affects the representation of software objects which it must maintain. Editors which follow the tree-editing or template-based paradigms commonly use abstract syntax trees (ASTs) to represent the objects under edit, and rely on an unparsing process to regenerate the concrete textual or graphical representation required for display. Such ASTs are very similar to the representations needed by back-end tools for semantic analysis of the objects concerned. In contrast, editors which follow a text-recognition paradigm, and allow essentially text editing operations on the displayed object, commonly use much more concrete syntax trees (CSTs) to represent the objects concerned. In subsequent discussions of integration paradigms, we therefore recognise the need for significantly different representations between the front-end and the back-end tools with which it is integrated.
Clearly, it can be argued that the choice of the specific architecture proposed above may bias

any conclusions about tool integration mechanisms which are arrived at by its use. A similar argument can be made about any architecture selected for experimental investigation. The argument should not, however, be seen as invalidating an experimental approach. Rather, it makes explicit the need to take account of such bias in presenting the experimental conclusions.

5 Tool integration paradigms

Software tools can be integrated by many different interfacing paradigms. In general, none of these paradigms is uniformly superior to the others. Each has its own advantages and disadvantages. Meyers [9] suggests that which one will prove most useful in the long run remains an open question.
We have investigated three different paradigms within the context of the architecture depicted in Figure 1, using the recognition editor UQ1. In this section, we introduce the paradigms concerned. We delay discussion of the strengths and weaknesses of each paradigm to section 8, where a comparative evaluation of all three paradigms is given.

5.1 The uncoupled paradigm

The simplest means of front-end/back-end "integration" is file and operating system based. We term this *the uncoupled interfacing paradigm* since all communication among tools is via operating system facilities and each tool has its separately executable code, with no direct dependence on the others existence. At present, many environments utilise this simple kind of integration.

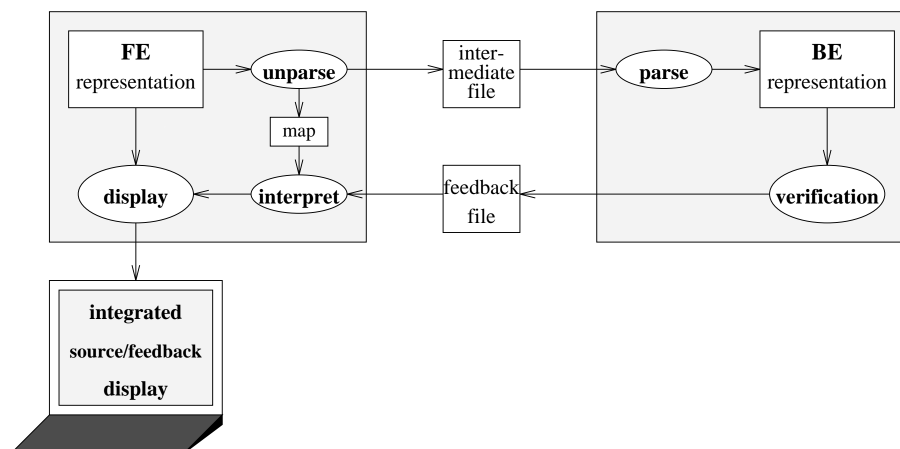

Figure 2: Information flow in the uncoupled paradigm

Figure 2 shows the typical information flow in the UQ1 realisation of the uncoupled paradigm. When verification of the document under edit is required, the front-end generates an intermediate ASCII file by unparsing its CST representation; this file is then parsed by the back-end to construct the AST representation required for verification. Feedback from the back-end is written to another intermediate file, and is integrated in the front-end display by a reverse mapping of the lines (and characters) positions in the first ASCII file to the CST representation from which it was produced.
The essential requirements for a generic realisation of such uncoupled tool integration are:

- a mechanism to specify the required content of the intermediate file, i.e. an unparsing schema,
- a user command to write the file and activate the back-end when verification is required, and
- a reverse mapping mechanism to interpret the feedback file produced.

5.2 The tightly-coupled paradigm

The directly contrasting approach to the uncoupled paradigm is the *tightly-coupled interfacing paradigm*, in which the tools are combined in a single executable system by linking compatibly compiled modules, communication among tools is by compiler generated calls, and they typically share an internal representation of the document concerned. Most experiments with such tightly-coupled integration appear in programming environments based on language-based editors such as the GANDALF project [11], the Cornell Synthesizer Generator [15], and the CENTAUR system [1].

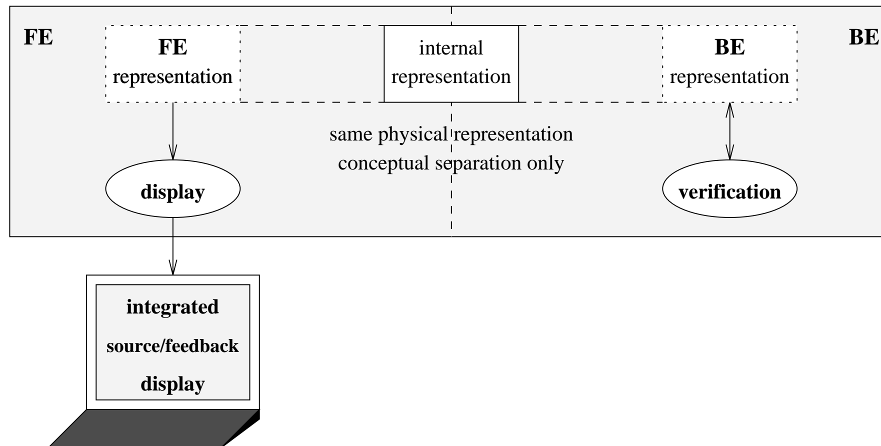

Figure 3: Information flow in the tightly-coupled paradigm

Figure 3 shows the typical information flow in UQ1's tightly-coupled paradigm. The front-end and back-end effectively share the same representation, with automatic integration of front-end and back-end displays. To support the tightly-coupled paradigm in a generic fashion, the UQ1 front-end editor exports to back-end tools an appropriate abstract syntax tree, as an abstract data type which

- is traversable by a standard set of tree navigation operations,
- allows decoration of its nodes with information resulting from such traversal, which can be interpreted and displayed by the front-end, and
- distinguishes those nodes or subtrees which have been changed since the previous tool activation to allow incremental processing when appropriate.

UQ1 retains explicit user activation of tightly-coupled tools, but the technique is easily adapted to automatic tool activation.

5.3 The loosely-coupled paradigm

As we will see in section 8, the uncoupled and tightly-coupled paradigms are two extremes for tool integration, with directly complementary advantages and disadvantages. Various compromises termed *loosely-coupled interfacing paradigms* can also be considered, in which tools communicate by some means smarter than basic operating system facilities, but each tool has sufficient separation to maintain system flexibility. There are two basic approaches to pursuing such a compromise: one is to use a purpose-built *database or object base* as in PCTE [2] and CAIS [10]; the other is to use *the message passing paradigm* as in Field [14] and SoftBench [4]. Some hybrid models combining these basic approaches have also been investigated [8, 12].

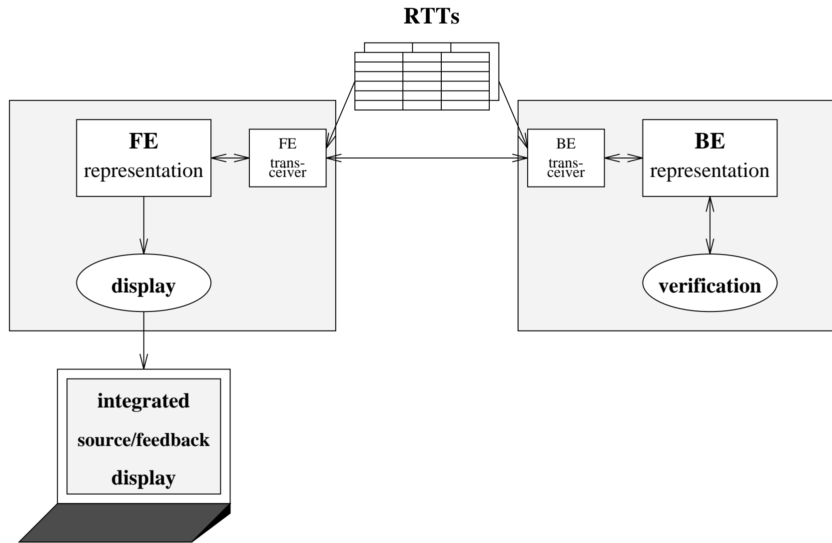

Figure 4: Information flow in the loosely-coupled paradigm

In UQ1, we have adopted a message passing approach for reasons similar to those asserted for Field in [14] but with more emphasis on fine-grained data integration. Figure 4 shows the typical information flow in UQ1's loosely-coupled paradigm. Front-end and back-end tools exist as separate, concurrently executing processes which communicate via some inter-process communication mechanism. Each maintains its own representation of the document under edit. Front-end and back-end representation transformation tables (RTTs) enable transceivers in the front-end and back-end to exchange AST/feedback transactions by direct communication when required. The UQ1 front-end controls this information flow by

- transmitting to the back-end an encoded AST representation of the source,
- transmitting requests for verification by the back-end, to be applied to some or all of the current representation,
- receiving requests from the back-end to adjust the current source display as feedback is generated, and
- signaling subsequent changes to the source in a way that minimises the communication costs involved, and enables the back-end tool to avoid re-processing unchanged source wherever possible.

Since the design issues arising in implementing this paradigm are more subtle than in the case of the uncoupled and tightly-coupled paradigms, we examine them more closely in the following subsections.

5.3.1 The representation transformation

The transformation from the front-end CST representation to the back-end AST representation may be seen as mapping a tree of CST node types to a tree of AST node types. In principle, the transformation may be arbitrarily complex but must have the following two properties.

Firstly, transmission of representation from the front-end to the back-end and transmission of displayable feedback from the back-end to the front-end imply that the transmitted representation must incorporate some frame of reference or *coordinates* in terms of which both ends can identify specific components. If we assume that the representation transformation is effectively made by the front-end, coordinates exchanged can only refer to AST nodes but not all node types are necessarily referenced. In effect, therefore, the coordinate node types must be some subset of the AST node types.

Secondly, all updates transmitted must necessarily be identifiable in the frame of reference. We note, therefore, that the node types used to define units of transmission must be some subset of the coordinate node types.

The granularity of incremental re-processing is necessarily determined by the back-end's needs and capabilities. The front-end must signal changes to the overall representation in consistent units. The granularity of update transmission must be no larger than that of semantic re-processing, otherwise retransmission of unchanged material may force the back-end to carry out unnecessary re-processing.

In principle, the granularity of update transmission should be as small as possible, to minimise transmission costs. In practice, the overheads associated with maintaining coordinate mappings may offset the benefits of very small transmission units.

Typically, the AST nonterminals required for semantic processing are normally a subset of those in the CST, or the grammar used to generate the CST can easily be adjusted to make it so. Similarly, the terminal or leaf nodes retained in the AST are a subset of those in the CST. In this common case, defining the transformation required involves identifying three subsets of the CST node types:

$$UnitNodeTypes \subseteq CoordinateNodeTypes \subseteq ASTNodeTypes \subseteq CSTNodeTypes$$

In this situation, the specification of the transformation needed is a simple adjunct to the grammar defining the concrete syntax, which identifies the CST node types included in the AST, frame of reference and transmission unit sets for each back-end tool.

5.3.2 Inter-tool communication strategies

The loosely-coupled paradigm involves front- and back-end processes communicating via an inter-process communication mechanism such as TCP/IP sockets, pipes, message queues or signals. Whatever mechanism is chosen involves some communication overheads. In considering the impact of such overhead, we must distinguish two possible user paradigms for the verification processing involved. One is where verification is automatic as user editing progresses, with "immediate" feedback of results. The other is where verification is performed only on demand. The choice between these paradigms is a significant user

interface issue, but has obvious implications for the inter-tool communication strategy — in general, automatic verification leaves less freedom of choice in this respect.
In practice, UQ1 is designed to provide verification feedback on demand, and our subsequent discussion examines two alternative strategies in this context. One of these, the so-called "eager" transmission strategy is equally appropriate when automatic verification is involved.

Incremental representation transmission

In transmitting document changes from the front-end to the back-end, there are two available strategies:

- *lazy transmission*, where all changed units are transmitted to the back-end just before the verification is invoked, and
- *eager transmission*, where after each user edit operation the changed unit or units are transmitted to the back-end immediately.

The lazy transmission mechanism enforces sequential processing by the front- and back-ends. Its response time to the user is obviously longer than that of the tightly-coupled paradigm because of the additional time of sequential transmission via relatively slow interprocess communication and because of the time consumed by the consequent AST update in the back-end.
In contrast, eager transmission distributes transmission to the completion of each edit operation so that it offers the potential for back-end AST update in parallel with further user edit activity. Eventually when back-end verification is required, all necessary transmission and update may be complete, so that verification can be carried out immediately.
To convince ourselves that the eager method will not affect or only slightly affect the user's normal edit operations in general, we consider two typical cases — initial program loading and a typical insertion operation.
A fully eager transmission strategy must transmit the entire program to the back-end tool as soon as it is loaded. Assuming that no buffering delays occur, the time to transmit the entire program when it is first read in should be small compared to the time needed to parse the input and build the CST. Even if programs are stored in some efficient parsed form, the transmit time should be no greater than the load time. Furthermore, the resultant delay experienced by the user must arise at some time, either at program loading with eager transmission, or when verification is first requested with lazy transmission.
When the user finishes an insertion operation, whether the additional delay caused by eager transmission is noticed by the user may depend on whether or not the editor is modeless. In a modal editor, such as UQ1, the user terminates insertion by an explicit key stroke or mouse click and then goes to the next operation. In this circumstance, a precious time gap exists which the front-end can use to transmit the changed unit(s) to the back-end.
In contrast, a modeless editor detects that the user is finished the current insertion only by the user's attempt to initiate the next operation. In this circumstance, eager transmission must contend with the user's need for immediate editor-response, and is therefore more likely to cause a noticeable delay.
Similar arguments apply to other circumstances in which eager transmission occurs.
In summary, we conclude that eager transmission can improve the observed response when verification is requested, without significantly degrading the editor performance at other times, particularly with modal editing. More detailed analysis is given in [25]. We have

therefore chosen eager transmission as the initial strategy to be used within our loosely-coupled interface.

Effective feedback processing

We must also consider the strategy for transmission of feedback by the back-end and its processing by the front-end. In practice, a choice of strategy exists which is analogous to that for AST transmission by the front-end.
With a *lazy* feedback strategy, no feedback is transmitted by the back-end until verification is complete, and no feedback is processed or displayed by the front-end until all feedback has been received. Compared to the tightly-coupled integration strategy, this necessarily causes an additional delay equivalent to the feedback transmission time.
With an *eager* feedback strategy, the front-end processes and displays each feedback message as it is received. In this case the additional delay noticeable by the user is no more than the time taken to transmit the first message. For some verification functions, of course, it is inappropriate to display feedback messages in the order which verification generates them, but in these cases the back-end tool can ensure that messages are transmitted in display order.
In summary, the eager approach to feedback processing seems preferable, in that it minimises noticeable delays due to transmission overheads without compromising the verification tool's ability to control feedback display. In practice, it is prudent to restrict the user from actions other than feedback browsing (or aborting verification!) while verification is in progress.

6 Experiments with tool integration

The uncoupled, tightly-coupled and loosely-coupled integration paradigms have been prototyped on UQ1. To investigate their relative effectiveness from the viewpoints of integration cost and runtime performance, we chose to integrate static semantic analysers for Pascal and Modula-2, and type checkers for Z, with UQ1-based editors for the languages concerned, using two or more of the paradigms in each case. In this section, we give some quantitative measures of the integration costs involved. In the following section, we give some measures of response times as observed by the user.
In general, to integrate an existing tool as an uncoupled back-end in UQ1, involves writing:

- an unparsing schema to define the intermediate file used, if necessary, and
- a shell script to encapsulate the tool activation, and transform its feedback output into a form suitable for interpretation by the front-end.

To integrate an existing tool as a tightly-coupled back-end in UQ1, is more difficult to characterise, as it depends on how easily the tool can be adapted to function by walking a tree of the form exported by UQ1. We can characterise this effort only by the total code rewriting involved.
To integrate an existing tree-based tool as a loosely-coupled back-end in UQ1, involves writing:

- a transformation description, as an adjunct to the editor's existing syntax description file, and
- a back-end transceiver to construct and maintain the tree involved, and to relay feedback to the front-end.

In general, we note that much of the back-end transceiver written in any back-end implementation language is likely to be reusable for other back-ends written in the same language, since the mechanics of decoding and reconstructing the encoded AST transmitted by the front-end is largely independent of its detailed structure.
For both tightly- and loosely-coupled tools, additional modification of the back-end may of course be appropriate to improve its incremental behaviour.
The following subsections summarise our experience with the Pascal, Modula-2 and Z tools, in these terms.

6.1 Integrating Pascal semantic analysers

Prior to our systematic study of integration paradigms, a tree-based incremental static semantic analyser for Pascal [7] had been derived from an existing Pascal compiler [21] and used as a tightly-coupled analyser in UQ1. This analyser consisted of around 10,000 lines of Modula-2 code, but the cost of its derivation from the original compiler was not accurately recorded.
This tightly-coupled analyser was subsequently adapted for loosely-coupled use as follows:

- Writing the transformation description file was a trivial task as the existing syntax was sympathetic. The resultant description was around 200 lines long.
- Developing the back-end transceiver involved around 700 lines of Modula-2 code (i.e., 7% of the size of the existing analyser).

The original Pascal compiler was also integrated as an uncoupled tool. Because the editor's existing unparsing for file output was acceptable to this compiler, no new unparsing schema was needed. Because the compiler's error output was equally compatible with the front-end editor's expectations for feedback, the script encapsulating the compiler's activation was only 11 lines long.

6.2 Integrating Modula-2 semantic analysers

An existing text-based Modula-2 semantic analyser, which consisted of 23,000 lines of Modula-2 code, was first adapted to provide a tightly-coupled, tree-based, non-incremental analyser. This involved reworking about 5,500 lines of code, or 23% size of the of the original analyser.
Adapting this for loosely-coupled use was then as follows:

- Writing the transformation description file was again a trivial task involving around 130 lines of description.
- Developing the back-end transceiver involved around 2,000 lines of Modula-2 code (i.e., 8% of the size of the existing analyser).

The original text-based analyser was also integrated as an uncoupled tool. Again, no new unparsing schema was needed, and the script encapsulating the analyser's activation was 5 lines long.

6.3 Integrating Z type checker

The two applications outlined in the previous subsections were relatively easy because of high compatibility between the front-end and back-end tools. To demonstrate integration in more hostile circumstances we also integrated two Z type-checkers, neither of which was amenable to tightly-coupled integration.
The first was Spivey's Fuzz tool [16], which as an existing binary application could only be integrated by the uncoupled approach. To do so involved the following:

- Writing the unparsing schema to produce Fuzz-compatible input was straightforward within UQ1's unparsing system, and involved a 260 line schema file.
- Writing an encapsulating shell script was relatively simple — a 30 line script was needed to transform the Fuzz error messages into a form acceptable to UQ1.

The second was Sufrin's Hippo [18], a Z type checker written in the functional programming language SML, and consisting of 2450 lines of code. This could not be integrated via the tightly-coupled paradigm, but because its source was available it was amenable to loosely-coupled integration. This integration was non-trivial because its internal representation was very different from that of the front-end and required complex transformation which the prototype loosely-coupled interface did not support. As a result this transformation had to be implemented by 1100 lines of SML code in the back-end itself. This work would be very much reduced if a more powerful generic representation transformation mechanism was available, ideally in the front-end transceiver.
Apart from this additional transformation code, integration effort was as follows:

- Writing the transformation description file to define the front-end extraction from concrete to abstract syntax took around 235 lines of description.
- Developing the back-end transceiver involved around 250 lines of SML code (i.e., about 10% of the size of the existing analyser).

7 Run-time performance of the integrated tools

To investigate the relative run-time performance of the integration paradigms, we measured CPU time consumption when static semantic analysis is invoked from the Pascal editor via each of the three analyser integrations outlined in subsection 6.1. The front-end editor used is identical in each case, and the back-end semantic analysers are derived from the same Pascal compiler [21].
Table 1 represents raw data from our first experiment and shows the CPU time consumed in direct response to specific user requests, using the three integration paradigms. The results are shown for six different Pascal programs, characterised by the number of lines of code (LOC) in each. All times are shown in seconds, measured to an accuracy of ten milliseconds on a Sun 4/75. Each specific time interval is measured by recording and subtracting two CPU running times immediately before and after the task.
The *initial loading* time is the CPU time consumed by the front-end editor in first loading the program from a text file. For all three paradigms this involves parsing the file concerned, but for the loosely-coupled paradigm it also involves eager transmission of an AST representation to the back-end analyser. (In all three cases, the actual delay observed by the user may be greater due to disc activity during loading.)

The *1st analysis* time is the CPU time consumed (by the front- and back-ends) in analysing the whole program. (In the uncoupled case, the actual delay observed by the user may be greater, due to disc activity involved in writing and reading the intermediate source and feedback files.) The *2nd analysis* time is the CPU time consumed for re-analysing the program without any intervening change, and distinguishes between incremental and non-incremental strategies in the analysers concerned.

LOC	interfacing paradigm	initial loading	loading ratio to TC	1st analysis	analysis ratio to TC	2rd analysis
50	UC	0.32	*1*	1.93	*40*	2.04
	TC	0.32		0.05		0.02
	LC modal	0.40	*1.3*	0.05	*1*	0.02
60	UC	0.37	*1*	1.99	*50*	1.99
	TC	0.37		0.04		0.02
	LC modal	0.47	*1.3*	0.04	*1*	0.02
490	UC	3.92	*1*	3.83	*11*	3.82
	TC	3.92		0.36		0.02
	LC modal	5.00	*1.3*	0.36	*1*	0.02
520	UC	4.07	*1*	4.10	*7.5*	4.18
	TC	4.07		0.55		0.02
	LC modal	5.30	*1.3*	0.55	*1*	0.02
900	UC	8.20	*1*	6.67	*6.2*	7.15
	TC	8.20		1.07		0.05
	LC modal	10.70	*1.3*	1.07	*1*	0.05
2200	UC	18.53	*1*	13.09	*5.6*	12.87
	TC	18.53		2.33		0.07
	LC modal	24.66	*1.3*	2.33	*1*	0.07

Table 1: The raw performance data

The results for the uncoupled and tightly-coupled paradigms show clearly the analysis performance disadvantage suffered by the uncoupled approach, and in particular its communication overheads and lack of incremental performance. The results for the modal eager loosely-coupled paradigm show a noticeable one third extra delay at initial load, but otherwise identical performance to that of the tightly-coupled paradigm. The two ratio columns in Table 1 demonstrate the consistency across a range of program sizes and types, except for uncoupled analysis of very small programs where initialisation costs have a dominant affect.

Given this consistency, we then focussed on a typical case for further comparison of the three integration paradigms, during editing operations. Table 2 represents a comparison for three interfacing paradigms which indicates the CPU time consumed by five specific user requests on the 520 LOC Pascal program. In addition to requests defined for Table 1, we have two extra requests. The *FE 5 changes* time is the CPU time consumed by the front-end for five user updates at five different points. The basic time (X_i) depends on the edit command and is irrelevant to our consideration. We note, however, that this time is increased by the time required for eager transmission of the resultant change in the loosely-coupled case. The *3rd analysis* time is the CPU time consumed by the front- and back-ends in re-analysing the changed program immediately after the fifth change. In each case the block enclosing the change made is around 50 LOC (the typical size of a procedure). Given the block-based incremental processing strategy of the analyser concerned [7], the five changes together imply re-analysing about half of the program.

interfacing paradigm	initial loading	1st analysis	2nd analysis	FE 5 changes	3rd analysis
UC	4.07	4.10	4.18	5*(X_i)	4.15
TC	4.07	0.55	0.02	5*(X_i)	0.25
LC modal	5.30	0.55	0.02	5*(X_i+0.08)	0.25

Table 2: The typical performance comparison

The uncoupled paradigm is again disadvantaged by its high overheads and lack of incremental reprocessing after the edit operations. More significantly, the results show that with the loosely-coupled paradigm an extra delay of less than 0.1 seconds occurs after each edit operation due to eager transmission of the change involved. Such a delay is barely perceptible to a user in any circumstance. Typically, a pause of at least this magnitude occurs between user operations, so in practice the eager transmission delay is likely to remain completely undetected. Otherwise, the eager loosely-coupled paradigm achieves analytic response identical to the tightly-coupled paradigm.

At the moment, the modeless eager and lazy loosely-coupled paradigms have not been implemented, but more precise data on front-end and back-end delays collected using the modal eager system, as detailed in [24], can be used in conjunction with the data in Table 2 to extrapolate an equivalent table for their performance. In Table 3 the times shown for the eager modeless and lazy approaches are extrapolated by adding the time consumed by back-end contributions, measured using the modal eager mechanism, to the front-end transmission and back-end analysis times shown for the modal eager mechanism. For example, in the case of the *3rd analysis* for the eager modeless paradigm, the figure 0.38 is derived from a transmission time of 0.08+0.05, where 0.08 and 0.05 refer to front-end and back-end delays respectively, as detailed in [24], and a back-end analysis time of 0.25.

interfacing paradigm	initial loading	loading ratio	1st analysis	analysis ratio	2nd analysis	FE 5 changes	3rd analysis	analysis ratio
UC	4.07		4.10	7.5	4.18	5*(X_i)	4.15	7.5
TC	4.07		0.55		0.02	5*(X_i)	0.25	
LC modal	5.30	*1.3*	0.55	*1*	0.02	5*(X_i+0.08)	0.25	*1*
LC modeless	**5.30**	*1.3*	**0.55**	*1*	**0.02**	**4*(X_i+0.08)+X_i**	**0.38**	*1.5*
LC lazy	**4.07**	*1*	**2.55**	*4.6*	**0.02**	**5*(X_i)**	**0.90**	*3.6*

Table 3: The extrapolated performance

Compared to the modal eager mechanism, the extrapolated results of the modeless eager mechanism differ in one minor way — the situation with initial loading is exactly the same, but the situation with editing is that the extra delay arising from eager transmission is now associated with the initiation of the next operation. Thus, the front-end extra delay for the fifth change is not included in the cumulative edit delays, but the corresponding front-end and back-end delays are added to the perceived time for the *3rd analysis* request. In magnitude, however, the extra delay concerned (0.13 seconds) remains barely noticeable. In contrast, the extra delays imposed by the lazy transmission strategy on both the first and third analyses are clearly noticeable, particularly for the first request, where the response time is nearly 5 times that available with the tightly-coupled paradigm.

In summary, our experiments confirm our hypothesis that the eager transmission strategy is preferable to lazy transmission, even when verification tools are activated by explicit user

request. If automatic incremental verification is required, eager transmission is inevitable.

8 An overall qualitative evaluation

In this section, we present an overall qualitative evaluation of each of the three paradigms, in terms of the user's and environment builder's criteria identified in sections 2 and 3.

8.1 The uncoupled paradigm

The obvious disadvantages of the uncoupled paradigm are from the user's viewpoint:

(a) *feedback behaviour*
A reasonable degree of feedback integration can be achieved, but limitations on the precision of CST/text/AST mapping achievable may be inherent in the intermediate and feedback file formats imposed by the verification tool.

(b) *response time*
The high overheads of unparsing, writing, reading and parsing the intermediate ASCII file dominate the response time achieved, and there is little or no potential for incremental re-verification.

The advantages of the uncoupled paradigm are from the system builder's viewpoint:

(c) *integration cost*
Where a default or existing unparsing schema is acceptable, the only effort required for integration is to encapsulate the tool involved by a shell script. In other cases, an unparsing schema may be required, but is typically easy to write using the unparsing schema capabilities used in most language-based editors. In summary, integration needs no programming effort, and is typically achieved at low cost.

(d) *independence between components*
The uncoupled tool interface allows wide divergence in the implementation technologies used in the front- and back-ends. Even a binary executable tool can be integrated.

8.2 The tightly-coupled paradigm

In contrast, the tightly-coupled paradigm shows the opposite characteristics. The advantages are from the user's viewpoint:

(a) *feedback behaviour*
Feedback from the back-end tool can be fully integrated with the front-end's primary document display at a granularity determined by the exported AST.

(b) *response time*
Since the front-end and back-end access (appropriate projections of) a shared tree structure, no communication overheads are involved. Since the back-end has full knowledge of the extent of change since the previous verification request, its potential for incremental reprocessing is not compromised.

The disadvantages of the tightly-coupled approach are from the system builder's viewpoint:

(c) *integration cost*
The cost of integration may be high, particularly when an existing tool has to be adapted to conform to the AST interface exported by the front-end.

(d) *independence between components*
Since the back-end is compiled or linked with the front-end to generate a single executable program, the front- and back-ends must be implemented using compatible technologies, i.e. mutually compilable languages.

8.3 The loosely-coupled paradigm

The loosely-coupled paradigm achieves most of the advantages of both the uncoupled and tightly-coupled paradigms, as follows:

(a) *feedback behaviour*
Feedback from the back-end tool can be fully integrated with the front-end's primary document display at a granularity determined by the AST involved – exactly as for the tightly-coupled paradigm.

(b) *response time*
Overheads imposed by data transfer are involved in this paradigm, but the experiment presented in Tables 2 and 3 demonstrates that the loosely-coupled paradigm typically offers the user a response time broadly comparable to that of the tightly-coupled case. The overheads are acceptably low, and incremental analysis is not compromised.

(c) *integration cost*
In general, the cost involved in loosely-coupled integration is no more and may be much less than that involved in tightly-coupled integration. The typical programming effort required is the development of a back-end communication program which was 700 lines of code in the case of the 10,000 line Pascal analyser. This compares favourably with the potential cost of adapting a tool for tightly-coupled use. The much higher cost incurred in integrating the Z type-checker Hippo reflects the inadequacy of the generic CST-to-AST transformation mechanism in the UQ1 prototype.

(d) *independence between components*
As with the uncoupled paradigm, the interface allows a wide divergence in the implementation technologies used in the front- and back-ends, since the back-end tool is only required to conform to the inter-process communication mechanism used. However, an existing binary executable tool cannot be directly integrated by this paradigm since some change to its I/O behaviour is required.

9 A more flexible tool integration architecture

From the qualitative evaluation given in the preceding section, a reasonable conclusion is that the loosely-coupled integration paradigm is superior to the uncoupled and tightly-coupled paradigms *in overall terms* in that it achieves most of the advantages of the other two. In particular cases, however, one of the other paradigms may still be needed — in integrating an existing binary-executable tool, for example, only the uncoupled paradigm can be considered.

This overall conclusion has been reached, however, in the context of a particular architecture being used in a particular way. In this section, we discuss a more flexible tool integration architecture, which avoids some of the limitations inherent in our prototype experiments.
In our prototype implementation, and our discussions to date, we have assumed that at most one loosely-coupled back-end tool is "active", i.e., in dialogue with the front-end editor, at any time. Maintaining this constraint when interleaved use of several tools is required implies an unacceptable loss of incrementality for each tool. If the user invokes tool B after tool A, tool A has to be killed, with the loss of all current intermediate results to let tool B be initialised and activated. Tool B then has to evaluate the entire document regardless of whether it was activated before. In this circumstance, the overall incrementality of the system is much reduced.
If, however, we relax this constraint by allowing the front-end to be in simultaneous dialogue with multiple back-end tools, the response times of the system during editing which we observed in section 7 are no longer guaranteed. With the eager transmission strategy, the front-end now has to update multiple back-ends at each operation, and the transmission overheads are proportional to the number of active tools involved.
In general, a software development environment may contain many loosely-coupled tools. The incremental potential of each tool should not be affected by the interleaved use of an unrelated tool. If the loosely-coupled mechanism is to remain an effective option, we must demonstrate that the basic advantages shown to date can be preserved in an architecture which supports multiple simultaneously active back-ends. In such an architecture:

- each tool may have its own AST requirement
- each tool may have its own transmission granularity

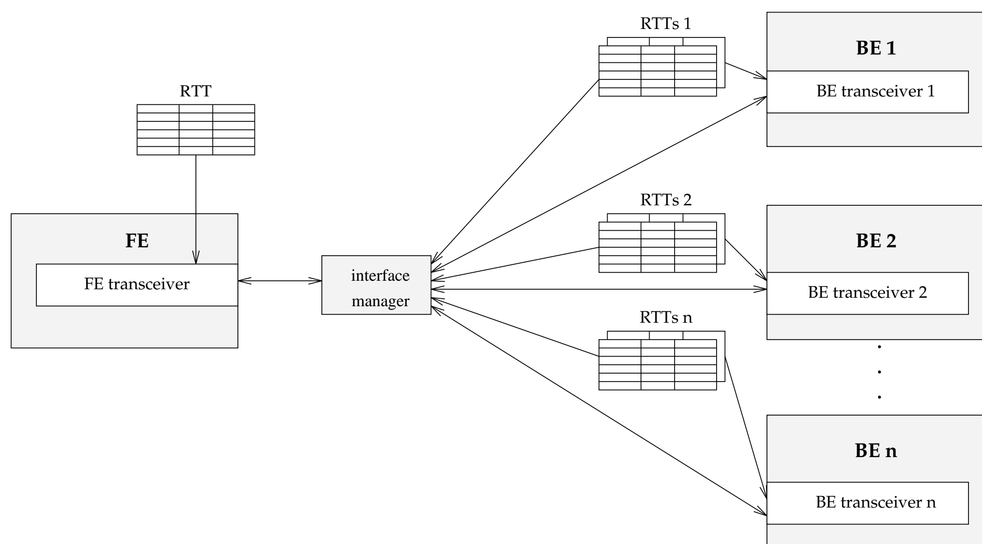

Figure 5: A generalised loosely-coupled architecture

To support multiple simultaneously active back-ends with the eager representation transmission mechanism, a generalised loosely-coupled interfacing architecture is proposed as shown in Figure 5. An *interface manager*, which the front-end sees as a single *loosely-coupled* tool, now plays the role of the loosely-coupled tool interface as far as the actual

loosely-coupled tools are concerned. The front-end transceiver maintains a single RTT, and eagerly transmits a single *generic* AST to the interface manager. The transmission delays incurred by the front-end in doing so are identical to those demonstrated for the single-tool case in section 7. From the viewpoint of the front-end and its embedded transceiver, the interface manager reduces the set of loosely-coupled tools active at any time to a single composite tool with a generic AST requirement.

The interface manager transforms and multicasts the generic AST transmitted by the front-end to the specific AST needed by each back-end as determined by the corresponding RTTs, and passes back-end feedback received to the front-end transceiver for the normal feedback processing. By operating asynchronously with the front-end, this service of multiple back-ends need not impact on the user's continued interaction with the front-end during editing. From the tool integration viewpoint, the interface manager is responsible for control and data integration between the front- and (loosely-coupled) back-ends. It can be seen as a control tool similar to the *message server* in Field [14] and SoftBench [6, 4]. In this context, a feasible option is also to locate the support of tightly-coupled and uncoupled tools in the interface manager. While this has the consequence of adding slight transmission overheads to the performance of tightly-coupled tools, it has the architectural advantage of unifying the front-end's view of tools, and simplifying its implementation. For uncoupled tools, the additional transmission overhead incurred is negligible compared with that inherent in the uncoupled interfacing paradigm itself.

Introduction of the loosely-coupled interface manager may be viewed as a fundamental change in the architecture introduced in Figure 1, bringing it closer to that of systems such as Multiview [8], which use a central AST-server to support both editing (front-end) and evaluation (back-end) tools. We note, however, that a recognition editor with a text-oriented interface like UQ1 or UQ2 is still likely to maintain its own document representation. We are currently investigating this AST-server architecture as the means of introducing *persistent* document manipulation to our recognition editors, and as a means of supporting *constructive* semantic tools, i.e., those whose semantic analysis of the document plays a determining role in its further development during input [20].

10 Conclusions

Integration of the tools used to prepare and verify software documents is vital to the effectiveness of the software development process. We have identified the key requirements of such integration from the viewpoints of both the tool users and the tool or environment builders. We have considered three possible integration paradigms, in the specific context of integrating semantic verification tools with a syntax-based document editor. Prototype implementation of all three paradigms in the same environment has enabled both qualitative and quantitative evaluation of their strengths and weaknesses, with respect to the requirements identified.

In overall terms, we see the loosely-coupled paradigm based on message-passing as the most effective, but note that particular circumstances may still dictate the use of other techniques. To achieve a workable environment which allows efficient interleaved use of multiple tools, we have proposed a modified architecture for implementation of the integration paradigms considered. We are now investigating the effectiveness of this architecture with respect to the introduction of document persistence to recognition editors, and to the integration of constructive semantic tools.

11 Acknowledgements

The development of concepts presented here, and the experimental work on which they are based, has been influenced by collaboration with our colleagues at the University of Queensland and elsewhere; these include Warwick Allison, David Carrington, Anthony Cheng, Wolfgang Emmerich, Jun Han, Ian Hayes, Derek Kiong, Chris Marlin, Mark Pedersen, Wilhelm Schäfer, and Andrew Wood. A research grant from the Australian Research Council has funded particular aspects of the work.

References

[1] P. Borras and D. Clément. CENTAUR: the system. *ACM SIGPLAN Notices*, 24(2):14–24, Feb. 1989. Also in *Proc. 3rd ACM SIGSOFT symposium on software development environments*, Nov. 1988.

[2] G. Boudier, F. Gallo, R. Minot, and I. Thomas. An overview of PCTE and PCTE+. *ACM SIGPLAN Notices*, 24(2):248–257, Feb. 1989. Also in *Proc. 3rd ACM SIGSOFT symposium on software development environments*, Nov. 1988.

[3] B. Broom, J. Welsh, and L. Wildman. UQ2: a multilingual document editor. In *Proc. 5th Australian Software Engineering Conference*, pages 289–294, Sydney, May 1990.

[4] M. Cagan. HP SoftBench: an architecture for a new generation of software tools. *Hewlett-Packard Journal*, 41(3):37–47, June 1990.

[5] D. Clément. A distributed architecture for programming environments. *ACM SIGPLAN Notices*, 26(1):11–21, Feb. 1991. Also in *Proc. 4th ACM SIGSOFT symposium on software development environments*, Dec. 1990.

[6] R. Ison. An experimental Ada programming support environment in the HP CASEdge framework. In F. Long, editor, *Proc. Int. Workshop on environments*, pages 179–193, Chinon, France, Sept. 1989. Springer-Verlag. Also in *Lecture Notes in Computer Science, Vol. 467.*

[7] D. Kiong and J. Welsh. Incremental semantic evaluation in language-based editors. *Software Practice & Experience*, 22(2):111–135, 1992.

[8] C. D. Marlin. A distributed implementation of a multiple view integrated software development environment. In *Proc. 5th. Conf. on knowledge-based software assistant*, pages 388–402, Liverpool, New York, Sept. 1990.

[9] S. Meyers. Difficulties in integrating multiview development systems. *IEEE Software*, 8(1):49–57, Jan. 1991.

[10] R. Munck, P. Oberndorf, E. Ploedereder, and R. Thall. An overview of DOD-STD-1838A (proposed), the common APSE interface set, revision A. *ACM SIGPLAN Notices*, 24(2):235–247, Feb. 1989. Also in *Proc. 3rd ACM SIGSOFT symposium on software development environments*, Nov. 1989.

[11] D. Notkin. The GANDALF project. *J. of systems and software*, 5(2):91–106, May 1985.

[12] H. Oliver. Adding control integration to PCTE. In A. Endres and H. Weber, editors, *Proc. European symposium on software development environments and CASE technology*, pages 69–80, Konigswinter, Germany, June 1991. Springer-Verlag. Also in *Lecture Notes in Computer Science, Vol. 509.*

[13] S. Reiss. Graphical program development with PECAN program development environments. *ACM SIGPLAN Notices*, 19(5):30–41, May 1984. Also in *Proc. 1st ACM SIGSOFT symposium on software development environments*, May 1984.

[14] S. Reiss. Interacting with the Field environment. *Software Practice & Experience*, 20(S1):89–115, June 1990.

[15] T. Reps. *Generating language-based environments.* MIT Press, 1984. PhD thesis – Cornell University, 1983.

[16] J. M. Spivey. The Fuzz manual. Internal document of the Programming Research Group, Oxford University, 1988.

[17] R. M. Stallman. EMACS: the extensible, customizable, self-documenting display editor. In D. R. Barstow, H. E. Shrobe, and E. Sandewall, editors, *Interactive programming environments*, pages 128–140. Mcgraw-Hill, 1984.

[18] B. Sufrin. Using the hippo system. Internal document of the Programming Research Group, Oxford University, June 1989.

[19] A. I. Wasserman. Tool integration in software engineering environments. In F. Long, editor, *Proc. Int. Workshop on environments*, pages 137–149, Chinon, France, Sept. 1989. Springer-Verlag. Also in *Lecture Notes in Computer Science, Vol. 467.*

[20] J. Welsh and J. Han. Software documents: concepts and tools. *Software — Concepts and Tools*, 1(1), 1994. To appear.

[21] J. Welsh and A. Hay. *A model implementation of standard Pascal.* Prentice-Hall, 1986.

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

[23] J. Welsh and Y. Yang. Tool integration techniques. In P. A. Bailes, editor, *Proc. 6th Australian Software Engineering Conference*, pages 405–418, Sydney, July 1991.

[24] Y. Yang. *Tool interfaces for software development.* PhD thesis, Department of Computer Science, The University of Queensland, Australia, June 1992.

[25] Y. Yang and J. Welsh. Software performance engineering – a case study for interactive software. In G. K. Gupta, G. Mohay, and R. Topor, editors, *Proc. 16th Australian Computer Science Conference*, pages 471–478, Brisbane, Feb. 1993.

[26] Y. Yang, J. Welsh, and W. Allison. Supporting multiple tool integration paradigms within a single environment. In *Proc. 6th International Workshop on CASE*, pages 364–374, Singapore, July 1993. IEEE Computer Society Press.