Software Documents: Concepts and Tools*

Jim Welsh and Jun Han

Software Verification Research Centre
The University of Queensland, Qld 4072, Australia

*A paper appeared in *Software — Concepts and Tools*, 15(1), pp. 12-25, 1994, Springer International/ACM; also appeared as Technical Report 93-23, Software Verification Research Centre, The University of Queensland, Brisbane, Australia, November 1993.

Software Documents: Concepts and Tools

Jim Welsh and Jun Han

Software Verification Research Centre
The University of Queensland, Australia 4072

Abstract

In this paper, we review software development as a document-based process, with the capture of a full but ideal development history as the assumed purpose of the documents concerned. We identify generic requirements for perusal, editing and verification of such documents, and illustrate how these requirements could be met in a software development environment based on current interaction technology. Finally, we propose a generic environment architecture for implementation of the facilities concerned, and outline how our own research work has addressed some of the requirements of this architecture.

1 Introduction

This paper starts from the basic premise that software development is a document-based process — each step in the development involves the preparation, manipulation or verification (in some sense) of one or more documents. The nature of the documents varies widely, from essentially informal documents, such as an initial requirement document, to highly technical documents expressed in formal languages, such as programs. The precise nature and range of documents involved depend on the methodology in use — an completely informal methodology may involve only informal text documents and program documents, a structured methodology may involve analysis and design documents in diagrammatic form, development by formal methods may involve formal specifications, refinements and proofs all in correspondingly formal, symbolic notations.

Whatever methodology is in use, significant tool support is required. The primary purpose of this paper is to examine the nature of software documents and of the tool support required, and to outline a developing architecture for the provision of such support.

In section 2 of this paper, we review software development as a document-based process, and the implications for capture, review and reuse of the development history as a set of software documents. In section 3 we review the form of software documents as they have developed over the years under changing technological constraints. In section 4 we illustrate how a document manipulation environment which meets the requirements identified in section 2 might appear to its users. In section 5 we present an overall architecture for implementation of such an environment. Finally, in section 6 we outline the approach being taken at the University of Queensland to develop such an environment. [1]

[1] Many of the ideas expressed in sections 2 and 3 have also appeared in [1], but are expanded and adapted here as an appropriate basis for the subsequent sections.

2 Software development as a document-based process

In considering the role of documents in software development, we assume that the end-product of a software development process is an *ideal development history* of the development, rather than just the compilable or executable programs that result. This view was implicit in the narrative presentations of systematic software development published by Dahl, Dijkstra and Hoare [2] and by Wirth [3] in the early seventies, which laid the foundations of structured programming. It was reinforced by Knuth's advocacy of "literate programming" [4] in the early eighties, and commended as sound engineering practice by Parnas and Clements in 1986 [5].

One simple interpretation of an ideal development history is as a purely informal account of the process which led to the design choices concerned. It is a history in the sense that it records the sequence of decisions taken in designing the program. It is ideal in that it need not necessarily record all of the mistakes and blind alleys pursued by the developers in arriving at the final design — Parnas and Clements called this "faking a rational design process". [2]

An alternative interpretation of the development history is as an explicit justification or *proof*, to a chosen degree of formality, that the program developed does in fact meet its requirements. In this case the history records not only the design decisions taken, but also the evidence that these decisions are appropriate in the context of the requirements that apply.

In the context of software development by fully formal methods, the concept of a development history must include the initial informal requirements document, the formal specification of the program concerned, all formal or informal design steps which advance its development from initial specification to final code, all proof obligations inherent in the design steps chosen, and the discharge of all these obligations in a theory appropriate to the specifications, programs, and languages concerned.

With less formal methods, the specification may remain informal, and the "proof" that the final product meets its requirements may rely on inspection and testing processes. In an effective development history, however, these processes and their outcomes must also be recorded.

Whatever methodology is used, the development history may attain a size and complexity which is orders of magnitude greater than the program it produces. In practice, the overall development history for a given product is usually captured as a *set of documents*. This partitioning meets both functional and managerial requirements, and often reflects the phases of development involved or the subdivision of tasks within a phase — one document will define the initial requirements, the next a more precise functional specification, and so on. It is important to realise, however, that the *relationships* between these documents, as well as the relationships between their components, form a vital part of the overall development history, and that the *traceability* of such relationships is a key attribute of the documents concerned.

Software development is therefore a process which involves the *preparation* and *verification* of the set of software documents which capture the development history of the software product concerned.

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

[2] In some cases, of course, it may be desirable to record such mistakes or blind alleys to warn subsequent maintainers of the dangers.

automatic generation of (parts of) the document by a computer-based tool.
By verification we mean any activity which establishes a document's consistency either with the methodological rules and constraints that apply or with the requirements of the software product's intended application. [3] Such activities may include inspection of the documents themselves, reasoning (either formally or informally) about their contents, and testing executable programs derived from them.
An effective environment for software development is one which supports the preparation and verification of software documents. In the following subsections, we review the preparation and verification processes in more detail, to determine the requirements for this support.

2.1 Perusing software documents

Preparation and verification of software documents are the primary goals of the software development process. However, many components of that process are carried out by human software developers, who require understanding of the documents produced so far. Reading and understanding existing software documents therefore takes up a major part of each developer's time. Because many of these documents are highly structured, and have many explicit and implicit relationships between them and their components, reading and understanding them is a nontrivial process, which can benefit significantly from computer-based support.
For offline reading of documents, the generation of hard-copy which is

- well formatted with respect to the documents' inherent structure, and
- cross-referenced with respect to the implicit and explicit relationships within them,

significantly aids understanding.
For online or interactive reading of documents,

- presenting documents as a sequence or hierarchy of meaningful views rather than as flat text,
- providing multiple alternative view formats for particular purposes,
- formatting each view with respect to its inherent structure,
- abstracting or suppressing detail to offset size limitations both in the screens used to display views and the brains used to perceive them,
- enabling navigation between views via the implicit and explicit relationships that exist, and
- enabling searching with respect to textual or semantic patterns,

all significantly aid understanding.
Because of the non-trivial nature of this understanding process, we use the term *software document perusal* to describe it. Effective support for document perusal is a major requirement of the software development process.

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

2.2 Editing software documents

Document preparation is commonly seen as an editing process, since the input of completely new documents is usually supported by the same tools as subsequent changes to such documents. Software is by definition amenable to change, and in practice most software systems, and hence the documents that express them, are subject to continuing change throughout their lifetime. To provide effective support for such change, we must analyse what is actually involved.

Consider some change which is required in a software product. The corresponding development history is, conceptually at least, a structure of design steps. In carrying out the change required, the first requirement is to identify the steps affected.

The strategy which identifies, traverses and edits the affected design steps is dependency-based — a change in one component requires a change in another if there is some dependency between the two. In general, such dependencies between software components arise in a number of ways, and these dependencies must be accessible to the developer via the editor if systematic change is to be achieved.

In carrying out each set of related changes, the developer wishes to make maximum use of the prior intellectual property inherent in the software being changed. No existing component which can be reused without change should have to be reconstructed, and adaptation of any component which needs to change should be achievable with minimum editing of its existing representation as perceived by the developer.

In achieving this latter goal, a particular issue is the *consistency strategy* enforced by the editing tools. In general, the components of a development history are subject to a variety of consistency constraints, syntactic, semantic and methodological, and the overall goal of development tools is to ensure that all such constraints are observed. To achieve this goal, a particular editing tool may adopt

- an error-preventing strategy, in which changes which violate the relevant constraints are simply precluded,
- an automatic error-signaling strategy, in which inconsistencies are allowed but immediately signaled to the developer, or
- an on-demand error-signaling strategy, in which inconsistencies are only signaled when the developer asks for them.

The simplest form of error prevention is a refusal to carry out an edit operation which introduces an error, but an alternative, more powerful, form is sometimes applicable. This involves *change propagation*, in which corresponding changes to other parts of the document are automatically carried out, when an edit operation that would otherwise create an inconsistency is requested.

Error-preventing strategies have the obvious appeal that the consistency of the software is assured at all times. If they are always enforced, however, changes to existing software sometimes require more than minimum effort from the developer's viewpoint. Automatic signaling of inconsistencies can also be a source of significant irritation to the developer in some cases.

Editing processes on software documents are often prolonged, and developers need protection against work loss through system failure or similar unexpected events during an editing session. For this reason *persistence* of edit operations at a granularity appropriate to the

developer's needs is an important characteristic of document editing. Conversely, of course, the developer must be able to undo any edit sequence when editing errors are made.
In summary, efficient and reliable editing of a software document requires

- a mechanism for identifying and tracing the dependency relations between components of the development history that determine which components are potentially affected by a change in a given component,
- an editing mechanism which allows changes to be made at relevant points in the development history without unnecessary reconstruction of any existing part of the history concerned,
- an overall consistency strategy which combines error-prevention (including change propagation when appropriate) with automatic and on-demand error-signaling, to achieve minimum overall effort and irritation from the developer's viewpoint, and
- controllable automatic persistence of edit operations at a granularity appropriate to the developer's needs.

These requirements reflect the editing needs of a single developer working on a particular document. In practice, of course, most software development is carried out by teams with team members working on separate but related tasks simultaneously. To ensure orderly editing of the documents concerned such a team also requires static and dynamic control of access to documents, and a version control mechanism to allow multiple versions of documents to exist when required.

2.3 Verifying software documents

Verification of software documents in our general sense may be 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.
Such tools may be applied in so-called *batch mode*, i.e., only 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 developer'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 is to ensure that all documents or parts of documents that need to be processed actually do get processed after any change. These complementary concerns are central to *configuration management*.
In all cases, effective support for tool-based verification requires that the developer has appropriate control over the tools concerned, that the documents concerned are made 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. Together, these issues define the concept of *tool integration* and in particular the integration of preparation and verification tools, which is again vital to the effectiveness of verification support for (document-based) software development.

In summary, therefore, whatever form of verification of software documents is involved, two generic requirements for verification tool support can be identified:

- an appropriate degree of integration between tools, particularly between preparation and verification tools, and
- a configuration management mechanism which ensures that, at an appropriate granularity, no more and no less than the necessary (re-)processing of software documents is carried out after any change.

2.4 Verification-directed editing

In the preceding sections, we have assumed that the developer is wholly responsible for all decisions which extend or amend a software document. Within this assumption, tools like pretty-printers or syntax-directed editors may relieve the developer of responsibility for determining low-level detail of the document's physical presentation, and incremental processing by verification tools may provide early feedback on changes that are made. The underlying abstract structure of the document, however, and the meaning thus implied, is wholly determined by the developer.

In more advanced forms of software engineering this is not necessarily the case. In these cases, computer-based verification tools may also play a significant role in determining parts of this abstract structure. Software development by formal methods is a typical case to consider.

To progress from an initial formal specification to a corresponding verified implementation, a systematic transformation process is often advocated in which the correctness of each transformation or refinement step is guaranteed, or verified before development continues. For development of software of any significant size or complexity, such a method is only feasible with the assistance of a refinement tool, which at each step identifies the possible applicable refinements, accepts the developer's choice from these, computes the refined components that arise, and generates the proof obligations necessary to ensure correctness of the refinement.

In fulfilling the proof obligations that arise, an interactive proof tool plays a similar role, accepting feasible proof steps from the developer as determined by the proof development to date, and generating the updated proof state, including further possible proof obligations.

Since the refinement and proof steps involved in such a process must be regarded as an integral part of the software's development history, the refinement and proof tools therefore play a significant role in determining the structure and significant content of the document which captures that history. In this sense they are *constructive* rather than purely *analytic* tools. The consequent question arising is how the contribution of such constructive tools should be represented in the documents concerned.

One option is to record only the developer inputs — since the constructive tools are presumably deterministic, the full development history can clearly be regenerated from this minimal information. As a perusable record of the development, however, the sequence of developer inputs by themselves are incomprehensible. Effective perusal can only be achieved in this case by involving the constructive tools in an animated replay of the original development, i.e., by actually *reliving* the development history concerned. Editing such a document would be achieved in a similar way, i.e., by reliving the history it contains up to some point, making some alternative developer contribution, and then reliving or replacing further parts of the original history as appropriate.

In practice, however, the responses from constructive tools, such as refinement and proof tools, are often computationally intensive and hence slow. The developer also has a natural need to look ahead in the existing history before committing to a particular change, which could necessitate multiple replays of parts of the developer input record, as editing progresses. For these reasons, this approach could lead to a fairly tedious editing process for the documents concerned.
The alternative option, of course, is to regard all contributions made by the constructive tools as an integral part of the development history, to be captured in the document concerned. In this case the document is a static record which is fully comprehensible for perusal purposes. When changes are required, the developer's paradigm is one of simply editing (relevant parts of) this record, and looking ahead or back is a natural part of the same paradigm. To achieve the same level of verification as during initial development, the constructive tools again have to reprocess the developer's changes incrementally, i.e, they must be tightly coupled to the editing process itself.
In summary, effective document capture of the software development history involving the use of constructive tools, requires a careful integration of these tools with the basic review and editing mechanisms for the documents concerned.

3 The form of documents

The role of software documents, in capturing a complete history of a development, has been recognised for some time. The form which documents have taken, and hence their effectiveness in fulfilling some aspects of their role, has been significantly constrained by the technology available for their support.
In the early seventies, interactive document manipulation was in its infancy. Simple line editors could be used to edit text files, from interactive teletypes or CRT terminals of equivalent functionality and speed. In this limited technological context, program specification and design evolved as paper-based processes, to be carried out prior to any interaction with the computer itself. Only the compilable code was held as a computer-based document. Documentation of each activity was seen to be important, but the forms taken by the documents corresponding to each phase were distinct. Graphical notations such as data flow diagrams, structure charts and control flow charts, were adopted for analysis, high- and low-level design. Between them, these documents could be seen as a form of development history, though in practice they often concentrated on the functionality involved rather than the reasons for a particular design choice.
Without adequate computer-based assistance, and because of the disparate representations used in each, the necessary correspondence between these documents was primarily the developer's responsibility. For the same reasons, it was also tempting to change only the final code versions of the programs developed, leaving the analysis and design documents unchanged, to the long-term detriment of the software's understandability and maintainability.
With respect to the requirements identified in section 2, the tools available at that time provided minimal support for editing and verification of code documents, and no support at all for others.
By the early eighties, interactive computing via fast character-based terminals was well established, and relatively sophisticated computer-based facilities for (textual) document preparation were available. In this context the first significant example of computer-based

support for narrative design histories emerged. This was Knuth's WEB system in support of "literate programming" [4]. The quality of the typeset narrative produced by WEB compared favourably with those hand-set in works by Hoare and his colleagues in the early seventies, but the input representation required by WEB reflected the relatively limited input capabilities of computers at that time. To use WEB, the developer prepared a text file in which the narrative commentary and code were interleaved in the required narrative sequence, with embedded commands to indicate the required structural relationships between code fragments. To prepare or review this input file the developer used a normal text editor, with no specific support for tracing or checking the consistency of the structural links represented by the embedded commands. When a program was extracted from this WEB file for verification, i.e., to be compiled and tested, the compilers and debuggers produced feedback in terms unrelated to the structure of the WEB narrative text file. The developers themselves had to maintain a mental map between the narrative and verifiable programs.
In overall terms, therefore, the WEB system effectively exploited the relatively sophisticated capacity of computers at that time to produce typeset publication-quality documents, but the developer's interaction with a WEB program during its development was constrained by the relatively primitive interaction capabilities available. With respect to the requirements identified in section 2, therefore, WEB significantly assisted offline perusal of design and code documents, but offered no improvement in their online perusal or editing. As a document preparation tool, WEB's lack of integration with corresponding verification tools also created additional problems for the developer during verification.
Ironically, the WEB system's development and release coincided with a radical change in computer interaction technology — pointing devices, bitmapped displays and powerful graphical capabilities. The immediate beneficiaries, however, were the structured methodologies using graphical representations. CASE tools quickly emerged to provide computer-based facilities for the preparation and maintenance of dataflow diagrams, structure charts and other graphical representations used in these methodologies. These tools reduced the cost of preparing and, more dramatically, of maintaining the charts and diagrams involved. By doing so, they also reduce the temptation to maintain software at code level only, and thereby contribute to its long-term maintainability.
The separation of, and relatively informal relationship between, the representations used at each phase of development remains a problem, however. For this reason, the development of fully integrated CASE tool sets covering the entire development cycle has proved much more difficult than the provision of preparation and maintenance tools for each phase. This is as much a flaw in the methodologies which the CASE industry seeks to support, as a deficiency in the technology on which the tools are based.
With respect to the requirements identified in section 2, these CASE tools provide significant support for perusal and editing of the documents used at each stage of the methodologies they support. Insofar as some CASE tools provide partial code generation from design documents, they can also be seen as constructive tools. In general, however, they fail to support and enforce the necessary relationships between documents produced in successive phases, with a consequent lack of integration from the developer's viewpoint, and a consequent lack of assurance for the overall development process.
With current interaction technology, there is no reason why fully integrated, fully effective support for software development cannot be provided in a way which more closely reflects the ideal development history concept, and more fully meets the requirements identified in section 2. In the following sections we present an architecture within which such support might be realised, and a strategy for its implementation which exploits both generic

construction of perusal and editing facilities and reuse of existing verification tools.

4 The user's view of documents

In section 2 we identified the requirements for effective support of software development as a document-based process. In this section, we indicate how such support might be delivered from the viewpoint of an interactive developer. To do so we use a small, even trivial, example to illustrate some of the document views which the interactive developer might see, and the kinds of manipulation which might be applied to them.

Individual documents. The most basic views that the developer expects to see are views of the individual documents involved in the development process. Through these views, the software developer can peruse, edit and verify the documents separately.

The requirements document. For demonstration purposes, suppose that we are required to develop a program to calculate the greatest common divisors (GCDs) of a file of positive integer pairs. The requirements document for the development of this program can be presented in a textual form as follows:

GCD.req

1. The program calculates the GCDs of a file of positive integer pairs.

2. In the input file, each pair occupies one line with the two integers separated by spaces.

3. In the output file, each line contains a pair of positive integers and their GCD, all of which are embedded in a piece of narrative text.

Because of the small size of our example, the requirements document is trivial, comprising only three natural language paragraphs, each containing just one sentence. In general, a requirements document is a long document composed of many sections, and in turn each section may be composed of many paragraphs. The paragraphs are typically expressed informally in natural language but may use diagrams and tables to clarify their meaning. Because of their informal nature, the layout of textual paragraphs, diagrams and tables in a requirements document is essentially as its author determines, via whatever text-editing, word-processing, or diagram-drawing support may be available.
Although a requirements document is essentially informal, its section and paragraph structure, together with a systematic labelling convention such as the paragraph numbering illustrated here, does provide a framework for cross-referencing within the document and from other related documents, as we shall see in due course.

The specification document. In our example development methodology, we use the specification language Z to formalise the requirements. The formal, top-level specification

document for the GCD example can be presented as follows:

GCD.spec

1. In Z, we represent the input file with a sequence of positive integer pairs, and the output file with a sequence of positive integer triples.

S_GCDs

$i? : \mathrm{seq}(\mathbb{N}_1 \times \mathbb{N}_1)$

$o! : \mathrm{seq}((\mathbb{N}_1 \times \mathbb{N}_1) \times \mathbb{N}_1)$

$i? = \mathit{first} \circ o!$

$\forall k : 1 \,..\, \#o! \bullet$

$\quad \mathit{second}\ o!(k) \ \mathrm{div}\ \mathit{first}\ \mathit{first}\ o!(k)$

$\quad \mathit{second}\ o!(k) \ \mathrm{div}\ \mathit{second}\ \mathit{first}\ o!(k)$

$\quad \neg\ \exists z : \mathbb{N}_1 \bullet$

$\quad\quad z > \mathit{second}\ o!(k)$

$\quad\quad z \ \mathrm{div}\ \mathit{first}\ \mathit{first}\ o!(k)$

$\quad\quad z \ \mathrm{div}\ \mathit{second}\ \mathit{first}\ o!(k)$

In practice, the input and output files must conform to paragraphs 2 and 3 of the requirements document.

Again because of the small size of the example, the specification document is composed of a single Z paragraph (i.e., a Z schema *S_GCDs*) and a commentary text. Note that some requirements, such as the input and output layouts in this case, are not necessarily expressed in the formal notation, and are only reflected in the associated commentary.
In general, a specification document may involve many Z paragraphs and many pieces of related commentary text, which may again be divided into sections. Unlike the requirements document, however, the Z paragraphs in the specification document are expressed in a formal language, and as such are subject to automatic formatting and graphics enhancement without any effort on the part of the document's author.

The design document. In practice, expressing all aspects of the GCD program requirements in a single Z paragraph is not necessarily good specification style, since it fails to separate the distinct issues of the sequential input/output structures on the one hand, and the precise meaning of "greatest common divisor" on the other. As a design step, therefore, we isolate the calculation of the GCD of one pair of positive integers as a sub-task, which can be specified as a function *D_GCD* in Z. Using this function, the entire task can be specified as a simplified Z schema *D_GCDs*, whose primary concern is the sequential input/output

relationship required. The following is a presentation view for the design document:

GCD.design

1. The calculation of the GCD of one pair of positive integers is isolated as a sub-task, which can be specified as a function in Z:

$$D\_GCD : (\mathbb{N}_1 \times \mathbb{N}_1) \rightarrow \mathbb{N}_1$$

$$\begin{aligned}
&\forall\, a, b : \mathbb{N}_1 \times \mathbb{N}_1 \bullet \\
&\quad D_GCD(a, b) \text{ div } a \\
&\quad D_GCD(a, b) \text{ div } b \\
&\quad \neg\, \exists\, z : \mathbb{N}_1 \bullet z > D_GCD(a, b) \land z \text{ div } a \land z \text{ div } b
\end{aligned}$$

2. The specification of the entire task is as follows:

D_GCDs

$$i? : \text{seq}(\mathbb{N}_1 \times \mathbb{N}_1)$$
$$o! : \text{seq}((\mathbb{N}_1 \times \mathbb{N}_1) \times \mathbb{N}_1)$$

$$i? = first \circ o!$$
$$second\ o! = D\_GCD \circ (first \circ o!)$$

In practice, the input and output files must conform to paragraphs 2 and 3 of the requirements document.

Alternatively, we note that the two Z paragraphs could be presented in separate views. In general, as we will see shortly, a view may present only part of a document, not necessarily the entire document.

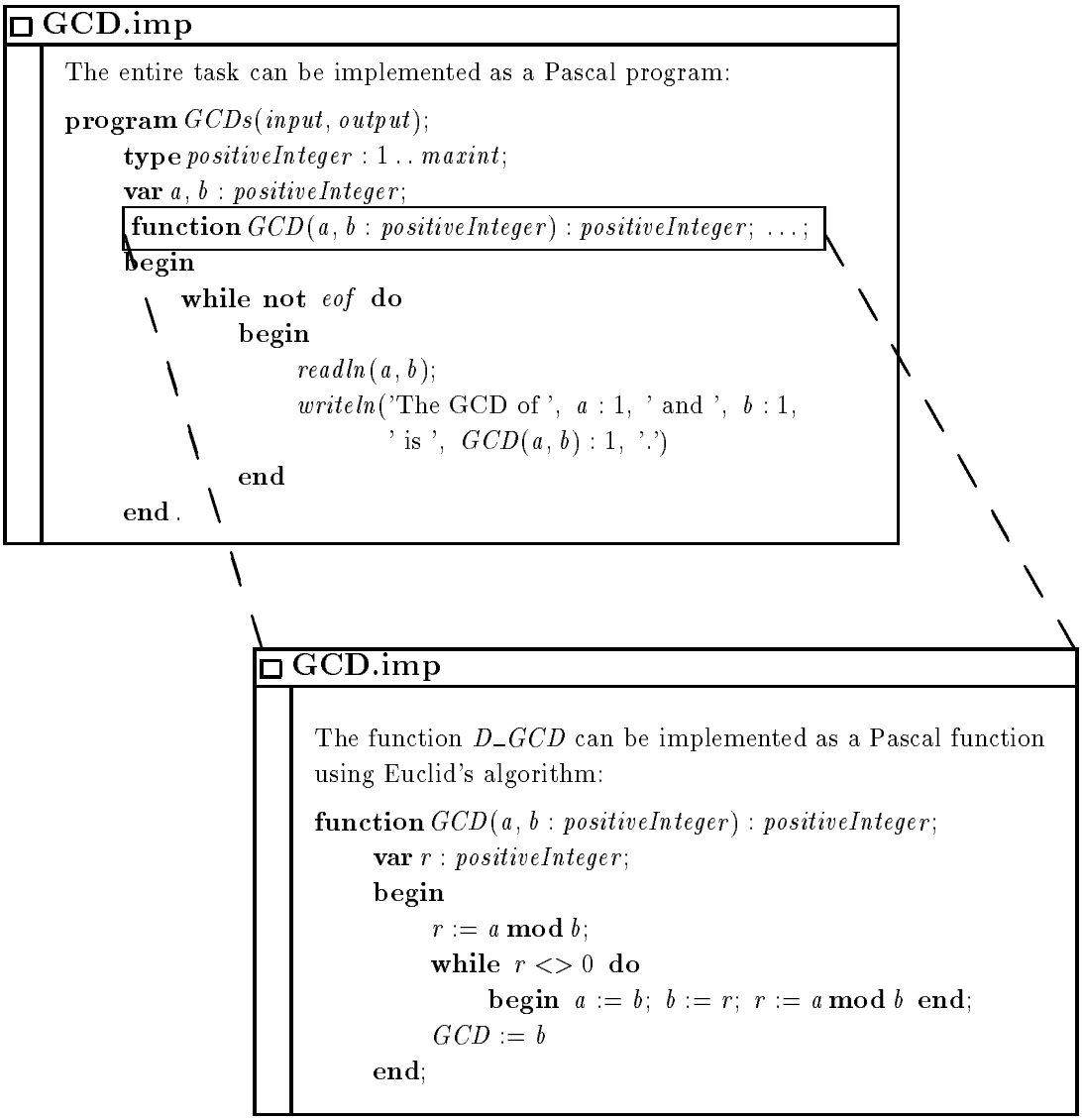
In principle, the above design step may need to be justified by discharging certain verification obligations, if a formal development methodology is being followed (see [6], for example). For simplicity, and without compromising the theme of this paper, we shall not consider this formal verification requirement here.

In general, the design of a software system may experience a sequence or hierarchy of such steps, and each step may focus on only part of the system. In our GCD example, however, we do not require further design steps, and we can go straight to implementation.

The implementation document. We choose Pascal as the implementation language. The function D_GCD can be implemented as a Pascal function GCD using Euclid's algorithm. Assuming that the positive integer pairs are read from the standard input stream and the GCDs calculated are written to the standard output, the entire task D_GCDs can be implemented as a Pascal program $GCDs$ with the nested Pascal function GCD.

We could present the implementation document in a single presentation view with the declaration for the GCD function being embedded in the overall program. When a program involves many function or procedure declarations, such a presentation view may quickly become incomprehensible. An alternative is apply abstraction or detail suppression by presenting the main program and the function or procedure declarations in separate views, with the nested function or procedure declarations being replaced by appropriate *handles* in the views for the enclosing constructs. According to this strategy, the implementation

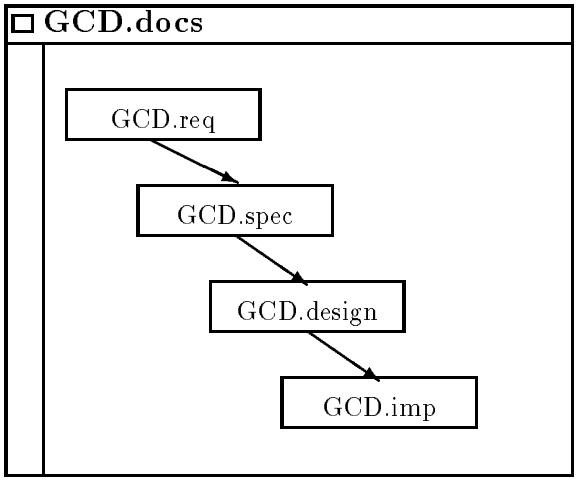
document for the GCD example can be presented in two related views:



In such a two-view presentation of the implementation document, the software developer can go from one view to the other view by navigating along the nesting relationship between them. In terms of the on-screen effect of such navigation, the target view can replace the existing view in the same presentation window or be presented in an additional, newly created window while retaining the existing view. This two-view presentation of the implementation document again demonstrates that a view may present only part of a document (in contrast to an entire document).

Documents overview and coarse-grained relationships. Another presentation view of particular interest to the software developer is a high-level overview of all the documents involved and the relationships between them, without the details of individual documents. This view can be most naturally presented in a graphical form. For the GCD example, this

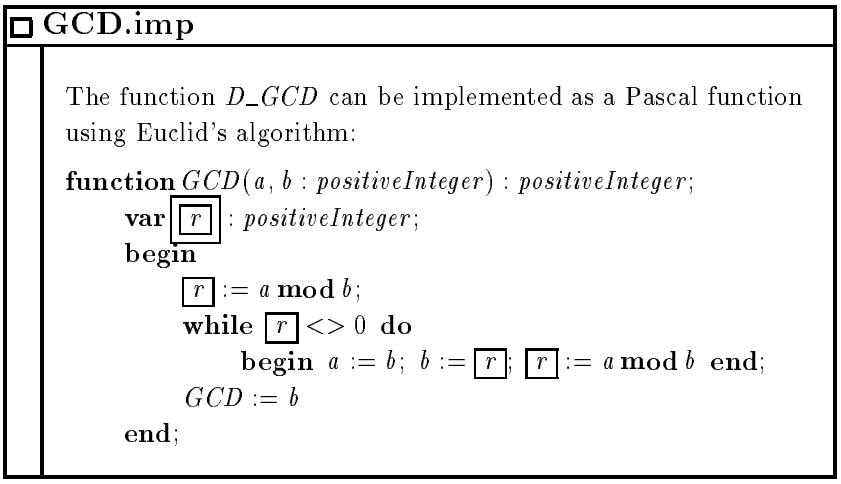
documents overview may look as follows:



where each box represents a document and each arrow represents a particular relationship between the documents connected. For example, the relationship between the requirements document and the specification document represents the fact that the specification document contains the top-level specification of the software system described in the requirements document.
Such relationships are inherent in the development methodology involved, but must be seen as part of the overall development history captured by the documents. For more complex software products the relational structure would not necessarily be linear — the high-level design phase for example may introduce many separate low-level design or implementation documents, each related to the same high-level design document. In general, effective presentation of the relational structure involved may require the application of a graph layout algorithm appropriate to the relations involved — the diagonal layout shown in our simple case is typical of the layout produced by such algorithms for a linear structure in a rectangular space.

Fine-grained relationships. In the above discussion, we have seen how relationships between entire documents can arise, and how they can be presented in a graphical view. As indicated in section 2, relationships between specific components of the documents can also arise and they must be presented in appropriate forms. In general, these fine-grained relationships can be divided into those between components of same document (intra-document relationships) and those between components of different documents (inter-document relationships).
Some fine-grained intra-document relationships can be presented in a hypertext-like manner within a textual view. When we want to know all the use occurrences of the variable r in the Pascal function GCD, for example, selecting the declaration occurrence of r may

automatically highlight all of its use occurrences:

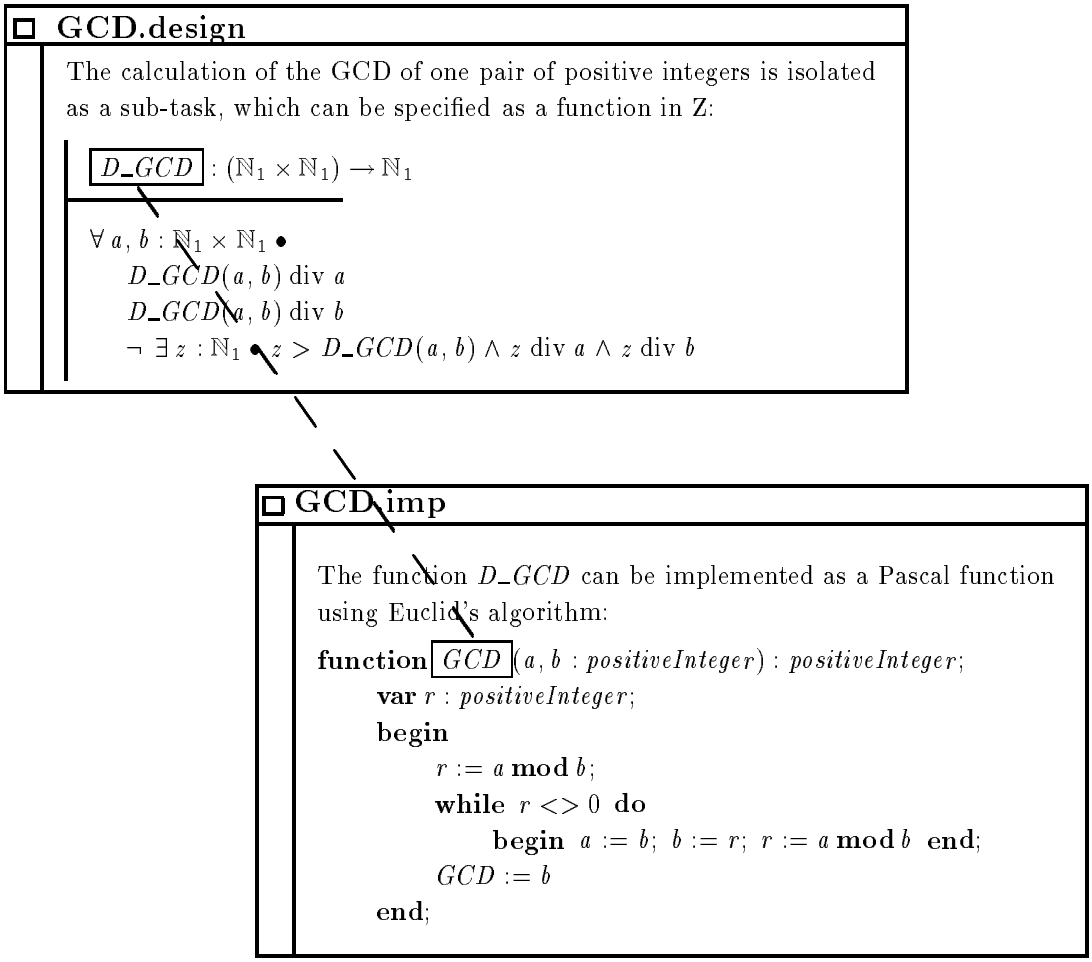


In this simple case, all these occurrences lie in the same view, but in general they might be spread over several views, in which case either the automatic creation of additional view windows to display these, or the presentation of a menu of relevant views for the user to choose from, are viable presentation options.

In general, each relationship between two components in a document offers a capacity to navigate from one of these components (as the focus of attention) to the other, with a consequent change of view if necessary.

As an example of fine-grained inter-document relationships, consider the "specified-by" relationship between the Pascal function GCD in the implementation document and the Z function D_GCD in the design document. By following this relationship, the software developer can navigate from a view of the implementation document which includes the function GCD to a view of the design document which includes the Z function D_GCD, or

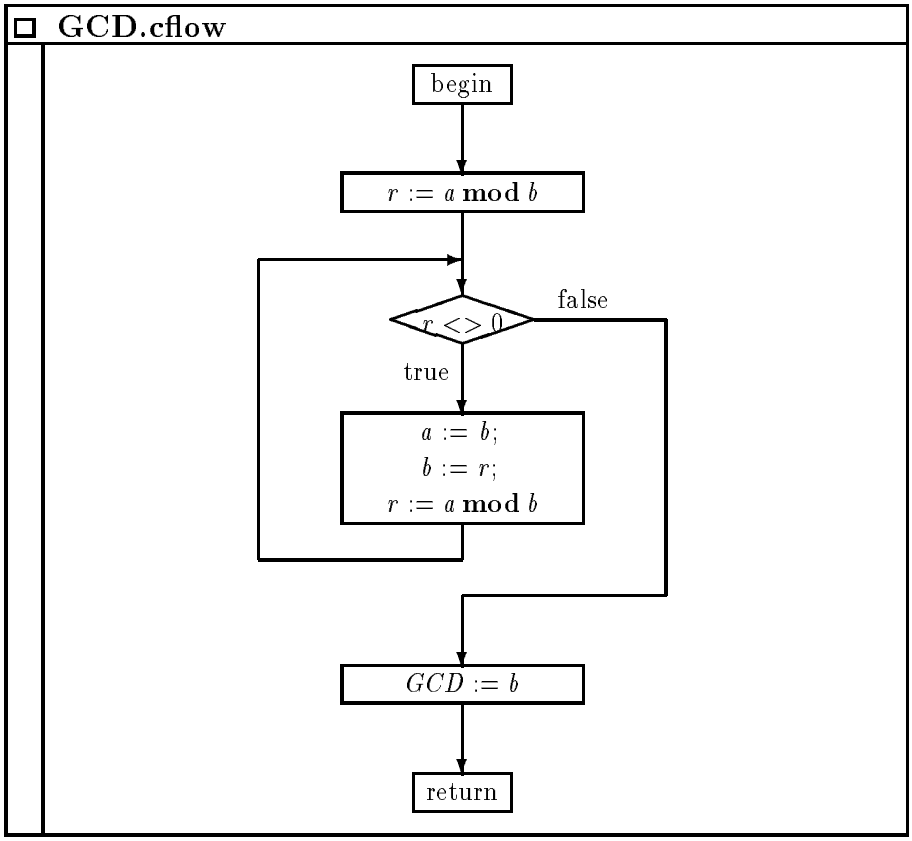
vice versa:



Similarly, reference relationships usually exist from components of a design or specification document to corresponding paragraphs or sections of the requirements document to achieve traceability between the corresponding development phases.
Note that inter- and intra-document relationships arise in different ways. Some, like the traceability relationships between a specification or design and the informal requirements document can only be introduced by the developers themselves. Others may be inherent in the methodology or language structures involved — a Modula-2 implementation module is necessarily related to its corresponding definition module, a proof obligation must eventually be linked to a corresponding proof which discharges it. Others, such as the relationship between an identifier use and its declaration, are more subtle and may be determined only by analytic tools which understand the scope rules of the language concerned.

Textual and graphical views. A presentation view can be a textual view or a graphical view. Most of the presentation views given above are textual views, but the overview of the set of documents which capture our GCD product is a graphical view which makes explicit the coarse-grained relationships between the documents.
Fine-grained intra- or inter-document relationships can also be handled in this way. In a typical Pascal program document, for example, the individual call relationships between procedure and function calls and their declarations can be explored by navigation or relative

highlighting, but the overall call pattern between the procedures and functions is commonly and best displayed in graphical form, as a *call graph* or *structure chart* view. Likewise the overall pattern of import relationships between separate module documents in a modular language like Modula-2 is often displayed in graphical form.
Our GCD example is too trivial to provide significant illustration of these graphical views, but an alternative example is a graphical view showing the control flow of the Pascal function GCD:



Semantically, this flow chart is completely equivalent to the textual representation of GCD's function body shown earlier. The example makes explicit that the software products we develop have an abstract semantic representation which is independent of the concrete syntax we use to denote them. For any component or aspect of the software, multiple equivalent concrete presentations are possible, any of which may be more suitable for particular users or their particular purposes at a given time. Some existing software environments, such as MultiView[7], are based explicitly on this principle, providing users with multiple simultaneous views of the same software components, and consistent manipulation across these.

Document perusal. The document views illustrated so far give a clear illustration of the features of software documents that have to be presented, but also of how online perusal of software documents can be supported. Of the requirements identified in section 2.1, only searching has not been illustrated, but this too can be modelled as navigation via a transient or virtual relation defined by the search pattern involved.
Support for offline perusal is in general achieved by adaptation of some of the online techniques illustrated — automatic formatting is much the same in either case, and cross-

referencing involves expressing the appropriate intra- and inter-document relationships in a form appropriate to hard-copy printing. Documents produced by WEB show clearly how this can be done[8, 9].

Document editing. In considering how document editing may be presented to the user, the first principle is that it cannot be separated from document perusal. The ability to move smoothly between perusal and editing activities is essential from the developer's viewpoint. Given the presentation characteristics illustrated for perusal, it is also clear that general-purpose text-editors and diagram editors cannot readily provide the facilities required. Knowledge of the nature of the documents concerned is necessary to effective support in this form. Language-based editors have been advocated as software development tools for some time, but in practice they have not achieved significant penetration in the marketplace. We believe this lack of success can be attributed to the nature and limitations of the editors offered so far, rather than any fundamental flaw in the concept.
In general, there are two major manipulation paradigms in language-based editors: tree-based manipulation and recognition-based direct manipulation. A comparison of these two paradigms can be found in [10]. It is now commonly recognised that users prefer system commands that represent *direct manipulation* of the displayed data representation to those whose effect is defined in terms of some abstraction of the display. To facilitate such direct manipulation of documents via language-based tools, syntactic recognition of the users' input and editing actions is clearly the preferred paradigm. Significant progress is being made with recognition editors for textual documents, but manipulation of explicit graphical representations or implicit relational structures within the same paradigm is an additional technical challenge.

Document verification. With language-based editors, syntactic verification of documents is automatically taken care of. Semantic verification processes may be supported in two ways. The first is automatic activation of appropriate semantic processes based on status or content change of documents. The second is to activate specific semantic processes only on an explicit request by the user. As discussed in section 2, the optimum is in general a mixed strategy. With explicit activation, however, we note that the interleaving of verification and preparation activities by the user may still be fine-grained, and the way in which control of verification processes is presented to the user must be closely integrated with the perusal and editing facilities. When viewing any software document, the user must have an easy means to check its verification status, and to invoke verification processing in direct relation to the view concerned.
By whatever means verification processes are activated, the feedback produced by them must also be closely integrated with the document views used for perusal and editing. Automatic display of all available feedback information is sometimes irritating, however, so users must again have control of when and which feedback is displayed, and the ability to navigate through documents on the basis of the existence of feedback of particular types at particular points.
With constructive verification tools, their control is necessarily integrated with the editing process itself, and the feedback they produce is directly incorporated in the document under edit.

5 An architecture for a document processing environment

In the previous sections, we have discussed how software development is seen as a document-based process and how the software developer might expect software documents, their relationships and their manipulation to be presented with computer-based support. In this section, we present an architecture for a document processing environment which will provide such support (see Figure 1).

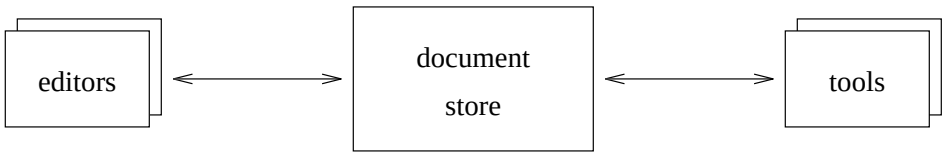


Figure 1: An environment architecture

The architecture comprises three kinds of component which we discuss in turn, before proposing an implementation strategy.

The document store. Central to our environment architecture is a persistent *document store*. It records and manages the internal representation of software documents in the environment. The persistent nature of the store facilitates the persistence requirement for edit operations that the user may apply through the views presented in the front-end *editors*, and the persistence requirement for document processing which relies on the back-end *tools*. The document store also provides a range of managerial features necessary to effective manipulation of software documents - static and dynamic access control, version control, and so on.
We see it as implicit in the issues and requirements discussed so far that the representation of documents maintained by the document store will be an abstract representation, in effect abstract syntax trees augmented by edges which represent the intra- and inter-document relationships concerned. These additional edges mean that the representations held may be thought of as an *abstract syntax graph (ASG)* which represents the entire set of documents capturing the development history of a product.
A range of arguments for this ASG representation can be found in [11]. In their simplest form, they can be reduced to two points:

- the user expects document persistence at the granularity of individual edit operations, and
- user navigation operations can span across documents.

Under these conditions, any persistent representation of documents which requires non-trivial transformation between the displayable, editable representation held within the editors and the stored persistent form (such as the parsing or unparsing of ASG segments to stored ASCII text representations) could not possibly meet the response-time expectations of the user for the operations concerned.

The front-end editors. The front-end editors present views of the software documents to the user, accept manipulation instructions from the user, and carry out view-level processing while document-level processing is communicated to the document store. These editors meet the perusal and editing requirements set out in the previous sections.

Given the wide range of documents that a developer may deal with within a single development, uniformity of the capabilities and user interfaces that these editors provide is a desirable characteristic for them.
Given that document preparation activities are often closely interleaved with verification activities, the editors should also provide, or be closely integrated with, the user interface for activation of verification tools and the presentation of feedback produced by these tools.

The back-end tools. The back-end tools are the tools which provide specific services for manipulating the software documents. These tools are usually semantic verification tools specific to the particular development methodology supported, and may include both analytic and constructive tools.
In general, these tools can be developed either separately from or within the framework of the environment architecture, but will all be integrated into the environment. Tools developed specifically for this environment architecture can fully exploit their integration with front-end perusal and editing facilities to provide better user control, incremental response and better feedback. In practice, however, many "off-the-shelf" tools can and should be utilised — existing type-checkers, compilers, and interpreters are all logical candidates for inclusion in this way. This diversity of tools requires that the environment architecture provide flexible tool integration capabilities.

An implementation strategy. To meet the requirement for user interface uniformity in perusing and editing different document types, and to deal with the complexity of developing a document store for each methodology and language set to be supported, an effective approach is via a *generic* implementation of the front-end editors and the persistent document store. This generic implementation is based on the general nature of software documents and their requirements for presentation, representation and manipulation, and is therefore independent of specific methodologies or projects.
To maintain appropriate flexibility with respect to existing tools, however, tool construction should not be seen as part of the generic environment. Instead, a generic interface for the integration of tools is envisaged. This interface must provide sufficiently close coupling to enable purpose-built tools to fully exploit the front-end and document store capabilities, but also have sufficiently loose coupling to enable existing off-the-shelf tools to be integrated without change.
To obtain a specific environment for a particular methodology or project, we then need to augment the generic front-end and document store with appropriate knowledge of

- the range of documents and document languages involved, in terms of their syntactic structure, the additional inter- and intra-document relationships to be supported, and any structural constraints that apply to these relationships;
- the views onto these documents that are to be supported by the front-end editors, in terms of their content and presentation;
- the range of specific manipulation operations to be supported for particular documents or their components, and any specialised interpretation needed of the generic perusal and editing operations that the front-end editors automatically support;
- the range of tools to be integrated in support of manipulation operations, together with the control and data integration conventions appropriate to each.

To specify these specific environment features, an *environment or document description language* is needed, based on the model of software documents underlying the generic implementation of the front-end editor and the persistent document store.

6 Work at the University of Queensland

Research into the construction of document-oriented software development tools has been in progress at the University of Queensland for some time. This started with the investigation of language-based editors which put specific emphasis on perusal support via structured displays, adaptive formatting, abstraction and detail suppression [12, 13, 14], and on editing support which combines the recognition editing paradigm with the reduced input effort and language help normally associated with tree-oriented editing [13, 15, 10].
Following early prototypes of the features, second phase investigations have significantly improved the display and parsing techniques used [16, 17], and have extended the editor's capabilities to handling multiple documents, each of which may be expressed in an interleaving of multiple languages [18, 19].
Related work has investigated the integration of verification tools with these recognition editors, using multiple tool integration paradigms to enable flexibility in tool integration while retaining the capacity for efficient, incremental tool processing when appropriate [20, 21, 22, 23, 24].
Further related work has examined the applicability of human-computer interaction concepts to the design of software development tools such as language-based editors, with a view to establish a more systematic approach to user interface issues inherent in these tools [25, 26, 27, 28].
Studies of the structures arising in formal methods activities such as program refinement, interactive theorem proving, and the overall formal development process have shown the feasibility, and the advantage, of capturing such development histories in a reviewable and reusable form [29, 30, 31]. Adaptation of the editor's tool integration facilities to provide prototype support for constructive tools has enabled theorem proving activities to be captured and reused in this way [32, 33].
Work is now in progress to extend the document concepts supported to include relational structures, enabling full capture, perusal and editing of the intra- and inter-document relationships in sets of documents. Document persistence will be achieved in this enhanced environment via a document store of persistent ASGs. Presentation of graphical as well as textual views of documents will be achieved using generic graph layout techniques. A central component of this current effort is the definition of an appropriate document description language as the primary input to a generic implementation of the facilities.

7 Conclusions

We have reviewed software development as a document-based process, with the capture of a full but ideal development history as the assumed purpose of the documents concerned. We have identified generic requirements for perusal, editing and verification of such documents, and illustrated how these requirements could be met in an environment based on current interaction technology. Finally, we have proposed a generic environment architecture for implementation of the facilities concerned, and have outlined how our own research work has addressed some of the requirements of this architecture.

Within these topics, we see the following as the most significant and challenging goals to pursue:

- recognition of inter- and intra-document relationships as first-class structural features in software documents,
- provision of appropriate navigational, editing and display capabilities based on textual, hypertextual and graphical presentation of such documents via a natural, direct manipulation interface;
- realisation of these features in a generic environment architecture which facilitates the provision of the necessary facilities for document storage, perusal and editing, while retaining the flexibility to exploit existing off-the-shelf tools for document verification.

These challenges will provide the basis of continuing research for some time to come.

8 Acknowledgements

The development of concepts presented here, and the experimental work on which they are based, have been influenced by collaboration with our colleagues at the University of Queensland and elsewhere; these include Warwick Allison, Paul Bakker, Brad Broom, David Carrington, Anthony Cheng, Wolfgang Emmerich, Ian Fogg, Ian Hayes, Dan Johnston, Cliff Jones, Derek Kiong, Melfyn Lloyd, Paul O'Keeffe, Gordon Rose, Wilhelm Schäfer, Cameron Strom, Mark Toleman, Luke Wildman, Andrew Wood, Mark Woodman and Yun Yang. Several research projects funded by the Australian Research Council have investigated particular aspects of the work.

References

[1] J. Welsh. Software is history! In A. W. Roscoe, editor, *A Classical Mind — Essays in Honour of C. A. R. Hoare*. Prentice-Hall International, 1994.

[2] E. W. Dijkstra O.-J. Dahl and C. A. R. Hoare. *Structured Programming.* Academic Press, London, 1972.

[3] N. Wirth. Program development by stepwise refinement. *Communications of ACM*, 14:221–227, 1971.

[4] D. E. Knuth. Literate programming. *The Computer Journal*, 27:77–111, 1984.

[5] D. L. Parnas and P. C. Clements. A rational design process: How and why to fake it. *IEEE Transactions on Software Engineering*, 12:251–257, 1986.

[6] S. King and I.H. Sorensen. From specifications, through design to code: A case study in refinement. In P.N. Scharbach, editor, *Formal Methods: Theory and Practice*, chapter 4, pages 103–137. BSP Professional Books, 1989.

[7] R. A. Altmann, A. N. Hawke, and C. D. Marlin. An integrated programming environment based on multiple concurrent views. *The Australian Computer Journal*, 20(2):65–72, May 1988.

[8] D. E. Knuth. *TEX: The Program.* Addison-Wesley, 1986.

[9] D. E. Knuth. *METAFONT: The Program.* Addison-Wesley, 1986.

[10] J. Welsh, B. Broom, and D. Kiong. A design rationale for a language-based editor. *Software—Practice and Experience*, 21(9):923–948, 1991.

[11] W. Emmerich, W. Schäfer, and J. Welsh. Databases for software engineering environments — the goal has not yet been attained. In I. Sommerville and M. Paul, editors, *Software Engineering — ESEC'93, Proc. 4th European Software Engineering Conference, Garmisch PartenKirchen, September 1993, LNCS 717*, pages 145–162. Springer, 1993.

[12] G. A. Rose and J. Welsh. Formatted programming languages. *Software—Practice and Experience*, 11:651–69, 1981.

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

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

[15] D. Carrington, I. Hayes, and J. Welsh. A syntax-directed editor for object-oriented specifications. In *Proc TOOLS Pacific 90*, pages 45–57, Sydney, November 1990.

[16] B. Broom and J. Welsh. Detail suppression systems for interactive program display. In *Proc 9th Australian Computer Science Conference*, pages 83–93, Canberra, January 1986.

[17] D. Kiong and J. Welsh. An incremental parser for language-based editors. In *Proc 9th Australian Computer Science Conference*, pages 107–118, Canberra, February 1986.

[18] B. Broom, J. Welsh, and L. Wildman. UQ2: a multilingual document editor. In *Proc 5th Australian Software Engineering Conference (ASWEC '90)*, pages 289–294, Sydney, May 1990.

[19] B. Broom, J. Welsh, and L. Wildman. A literate rigorous program case study. In *Proc 5th Australian Software Engineering Conference (ASWEC '90)*, pages 321–326, Sydney, May 1990.

[20] D. Kiong and J. Welsh. Incremental semantic analysis. In *Proc 10th Australian Computer Science Conference*, pages 126–136, Geelong, February 1987.

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

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

[23] J. Welsh and Y. Yang. A loosely-coupled tool interface for interactive software development. In *Proc 15th Australian Computer Science Conference*, pages 967–980, Hobart, January 1992.

[24] Y. Yang, J. Welsh, and W. Allison. Supporting multiple tool integration paradigms within a single environment. In H.-Y. Lee, T. F. Reid, and S. Jarzabek, editors, *Proc. of the 6th International Workshop on Computer-Aided Software Engineering (CASE'93), Singapore, July 1993*, pages 364–374. IEEE Computer Society Press, 1993.

[25] J. Welsh and M. Toleman. A case study in user interface design. In *Proceedings HCI Australia '89, Melbourne*, pages 19–28, November 1989.

[26] M. A. Toleman and J. Welsh. Retrospective application of user interface design guidelines: A case study of a language-based editor. In J. H. Hammond, R. R. Hall, and I. Kaplan, editors, *People Before Technology—Proc OZCHI'91*, pages 33–38, Sydney, November 1991.

[27] J. Welsh and M. Toleman. Conceptual issues in language-based editor design. *International Journal of Man-Machine Studies*, 37:419–430, 1992.

[28] M. A. Toleman, J. Welsh, and A. J. Chapman. An empirical investigation of menu design in language-based editors. In H. Weber, editor, *Proc. of the 5th Symposium on Software Development Environments*, pages 41–46. ACM Press, December 1992.

[29] J. Han and J. Welsh. User interface requirements for interactive verification systems. In *Proc 5th Australian Software Engineering Conference (ASWEC '90)*, pages 253–258, Sydney, May 1990.

[30] J. Han and J. Welsh. Structural representation of proofs. In *Proc 6th Australian Software Engineering Conference (ASWEC '91)*, pages 163–176, Sydney, July 1991.

[31] J. Han and J. Welsh. Object organisation in software environments for formal methods. In M.-C. Gaudel and J.-P. Jouannaud, editors, *Proc. of the 4th International Joint Conference on the Theory and Practice of Software Development (TAPSOFT '93), Orsay, France, April 1993, LNCS vol 668*, pages 299–313. Springer, 1993.

[32] A.S.K. Cheng, J. Han, J. Welsh, and A. Wood. Providing user-oriented support for software development by formal methods. In H.-Y. Lee, T. F. Reid, and S. Jarzabek, editors, *Proc. of the 6th International Workshop on Computer-Aided Software Engineering (CASE'93), Singapore, July 1993*, pages 156–165. IEEE Computer Society Press, 1993.

[33] A.S.K. Cheng, J. Han, J. Welsh, and A. Wood. Incorporating constructive tools into a generic language-based editor. In *Proc. 7th Australian Software Engineering Conference, Sydney, September 1993*, pages 197–208. Institution of Radio and Electronics Engineers Australia, 1993.