SOFTWARE VERIFICATION RESEARCH CENTRE

DEPARTMENT OF COMPUTER SCIENCE

THE UNIVERSITY OF QUEENSLAND

Queensland 4072
Australia

TECHNICAL REPORT

No. 94-37

Relational Views for Program Comprehension

Tim Jones, Warwick Allison, David Carrington

October 1994

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

Note: Most SVRC technical reports are available via anonymous ftp, from ftp.cs.uq.edu.au in the directory /pub/SVRC/techreports.

Relational Views for Program Comprehension

Tim Jones, Warwick Allison, David Carrington
{tsj, warwick, davec}@cs.uq.oz.au

Software Verification Research Centre
Department of Computer Science
The University of Queensland
Queensland, Australia 4072

TR 94-37

Abstract

In this paper we describe UQ✯, an integrated development environment that is currently under construction at the University of Queensland. Its architecture supports the definition of multiple documents and multiple document types, and allows the relationships that are implicit within the set of documents to be represented explicitly. We identify two techniques that aid program comprehension which require knowledge about the relationships that exist in and between documents. They are program dependency analysis and literate programming. Two simple examples are presented to illustrate the flexible definition of relations within such an architecture and the use of relations for presentation of, and navigation through, various views of a program and its related documentation. These examples highlight the application of such an approach to program dependency analysis and literate programming.

1 Introduction

Program comprehension, also referred to as program understanding, is the act of perceiving the meaning and structure of a program. When is program comprehension important? Software maintenance takes over 50% of the total expenditure that is allocated to a software system during its lifetime and program comprehension takes at least 50% of the time spent on the maintenance task [11, 12]. However, program comprehension is not solely a maintenance issue. Program comprehension is also a significant task during implementation. For example, individual members of programming teams continually review each others' code for both quality and corrective reasons. Even individual programmers reviewing their own code that was written over an extensive period of time will, for various reasons, find it necessary to investigate what various parts of their program do. Additionally, program comprehension is important during testing and debugging. Programmers must understand programs to devise complete and comprehensive test cases and to locate bugs. The need for tools to support program comprehension is well documented [19]. These tools should support program comprehension during implementation, maintenance, testing and debugging.

Wilde [32] emphasises that "*A key to program understanding is unravelling the interrelationships of program components*". In the context of a program source document, these relationships are more commonly known as program dependencies. Since the late seventies, much work has gone into the theoretical aspects of program dependencies [35, 34, 27, 28, 15, 24, 32, 21, 14] to pave the way for automating their extraction and presentation. This work has resulted in a wide variety of tools that address this issue. Large integrated environments such as Pecan [22], PV [7], MicroScope [1], ProDag in the Arcadia environment [23] all provide views of a limited set of program dependencies for a specific implementation language. Stand-alone program analysis and editing tools such as PUNS [12], Whorf [3], DgQuery and its associated tool-set [33], CIA [9] [10] and CIA++ [8] provide more comprehensive coverage of program dependencies, but are still language specific. Some program analysis tools such as LogiScope [18] are capable of analysing many different languages, but only provide views of a limited set of program dependencies.

Another important key to program comprehension is the documentation that accompanies the program. One well-known approach is the literate programming style [16]. However, Knuth's literate programming system WEB fell short of an ideal environment to encourage good documentation practice. Broom [5] improved on WEB by providing an interactive on-line presentation and simultaneous manipulation of both documentation and code as well as off-line presentation of a 'literate program' in an editor/browser called Sue. However interactive systems that support literate programming should not stop at this. Tools that support design and reverse engineering utilize graphical diagrams to convey ideas about a program. Literate programming and environments that support it should include appropriate graphical notations to allow enriched explanations of programs.

The comment by Teitelbaum that "*Programs are not text; they are hierarchical compositions of computational structures and should be edited, executed, and debugged in an environment*

that consistently acknowledges and reinforces this viewpoint" [26] indicates the need to integrate programming tools into a single environment based on a uniform representation. This comment was associated with an environment based on a structure editor which emphasises a fixed view of the program. Welsh [29] indicates that environments not need be restricted to structural views to reinforce the fact that programs are hierarchical compositions of computational structures. Ambras extends this concept and indicates that "*Programming environments that support evolutionary software development must include tools that help programmers understand complex programs*" [1]. We believe that these ideas should be extended further so that the documentation associated with the design and development of a program's source code is manipulated in the same way as the program's source code. Thus a programming environment should reinforce the fact that programs and their associated documentation are hierarchical compositions and should provide facilities to edit, execute, debug and understand them.

In order to support these concepts, we are developing an environment, UQ✯, whose architecture supports the definition of multiple documents and multiple document types, and allows the relationships that are implicit within the set of documents to be represented explicitly. The documents and views supported by such an architecture may be textual, graphical or both. The aim of this approach is to provide the ability to incorporate documents from the earlier phases of the software development life-cycle into the same architecture as the program, and to emphasise the inter- and intra-document relationships that allow more flexibility in defining and navigating through views of the software system than do conventional editors and environments. An important issue in the conceptual design of this architecture is that the program source is just another document whose language happens to be compilable into executable form. UQ✯ provides the flexibility to define views of program dependencies through the definition, manipulation and use of relations. Furthermore it encourages literate programming. Programs which are not literate and their development documentation can be loaded into the environment and woven together by creating the appropriate relations as the programmer explores the documents.

There are three accepted categories of theories that describe the cognitive processes involved in program comprehension. Corbi [13] describes these as the *bottom-up*, the *top-down* and the *opportunistic* theories. Bottom-up theories are based on the notion that a programmer understands a program by iteratively abstracting and connecting together 'chunks' of code. Top-down theories are based on the notion that the programmer uses their own experience and attempts to confirm their expectations. Opportunistic theories are a mixture of the first two types of theories, where the programmer uses an *as-needed* rather than a systematic approach to understanding the actual code [17]. By supporting flexible views of program dependencies and the ability to peruse development documentation in a 'literate' style, UQ✯ will support all three types of theories, particularly opportunistic theories.

In this paper, we discuss in more depth the importance of relations in program comprehension in section 2, and in the following sections illustrate how UQ✯ takes advantage of this to provide an integrated development environment that addresses these

issues. In section 3, an overview of the architecture of UQ✯ is presented. An example is presented in section 4, showing the derivation and application of relations for program dependencies, and section 5 contains another example which shows the use of relations for navigation within and between documents. Section 6 presents an overview of the work presented in this paper and related work at the University of Queensland.

2 Relational Views for Program Comprehension

In this section, we extend the ideas presented in the introduction to show the important role that relations play in program comprehension. By treating relations as an underlying conceptual structure of a programming environment, all the views discussed in this section are available to the programmer. We confine the discussion to two areas in which relations form the foundation of views that contribute to program comprehension. The first area is program dependencies and the second area is relationships between development documents and program source documents.

2.1 Program Dependencies

Program dependencies are relationships between a set of program elements that are dependent either syntactically or semantically on another set of program elements. In an environment based on relational structures, these relationships can be made explicit within the structure in which the program source document is contained. If inter- and intra-document relations are handled in the same fashion, the expression of program dependence relations is not restricted to being within a single program source document; they can be expressed between program source documents of both the same and different languages. This means that program dependencies can be traced through sub-programs that are written in different languages from the parent program under investigation.

Discussion of program dependencies in this section is limited to those found in conventional imperative programming languages. When considering such languages, we identify four classes of program dependencies: data-type dependencies; data-item dependencies; procedure/function dependencies; and module dependencies. Each of these is presented individually in the following sub-sections.

2.1.1 Data-Type Dependencies

In typed programming languages, data-type dependencies are the simplest type of dependency. Such languages have a finite set of pre-defined types; however in most of these languages the programmer can use these basic types as building blocks to create new, more complex types. Thus a programmer who has to understand a program, at some stage will be required to work out how these more complex types are constructed.

2.1.2 Data Item Dependencies

Data items are any components of a language that represent a particular value. Examples of

data items are variables and constants. In most languages, data items are dependent on two things: their declaration which specifies what type of value can be stored; and other data items that are used to derive their value. In the first case, the programmer is interested in the type of a given data item. This type may be a complex type that requires further investigation (section 2.1.1). In the second case, the programmer is interested in the effects of operations on data items. There are two questions to consider in this case:

1. What operations affect a particular data item at a given place in the program?
2. What data items are affected by a change in a given operation?

The first question involves isolating the statements that have a direct effect on the given data item. Weiser [27] described a related technique, which he called *program slicing*, as a method used by experienced programmers for abstracting and understanding programs. This technique is also an excellent and widely used technique for debugging programs [28]. A program slice is formally defined as the minimal subset of a program that produces a selected subset of the program's original behaviour. Weiser [27] says, in general, automatically finding a slice is impossible. However dataflow algorithms can be used to approximate a slice where the behaviour subset is the values of a set of variables at a statement.

The second question involves isolating statements which use the result of the given data item to derive the values of other data items. Yau [35] describes this as *ripple effect analysis*. This technique is most applicable to maintenance [35, 34], and like program slicing, it relies on dataflow algorithms to locate the desired information.

2.1.3 Procedure / Function Dependencies

Procedures and functions[1] can be seen as functional black boxes, providing their full behaviour is known and does not need to be changed. However if this is not the case, programmers trying to understand various aspects of a function's behaviour are faced with the problem that a function may contain references to program elements that are not defined in the function. Such references are known as a function's global dependencies. In structured languages like C, Pascal and Modula-2, the global dependencies of a function may be any of the following:

- Global type declarations;
- Global variables;
- Other functions; and
- Constants.

As with data item dependencies, a programmer may be interested in the declaration of the dependency, or the operational effects of the dependency.

1. In this section the word function is used to mean procedure as well as function.

2.1.4 Module Dependencies

In this paper we consider modules to be separate files such as in Modula-2. By this definition, classes in object-oriented languages such as Eiffel can also be seen as modules. As with functions, modules can contain references to program elements that are not defined in the module. Such references are known as the module's global dependencies. In structured languages, the global dependencies of a module may be any of the following:

- Global type declarations;
- Global variables;
- Global functions; and
- Constants.

The global dependencies of a module are always contained in another module, but are not necessarily the entire module. Thus this type of dependency is a more abstract dependency than the ones presented in previous sections. As well as module dependencies, a programmer may be interested in the declaration of the dependency, or the operational effects of the dependency.

2.2 Relations Between Development and Program Documents

Development documents reflect both functional and structural aspects of a program's implementation, as well as reasons for design decisions. Typical development documents include requirement, specification, design and program description documents. These documents may be textual, graphical or both. Advocates of top-down program comprehension theories indicate that domain knowledge is the fundamental starting point for program comprehension. In many cases, particularly program maintenance, programmers will have a very limited knowledge of the domain. In such cases, programmers rely on the development documentation to gain this knowledge. This task is potentially time-consuming since the set of such documents associated with a program is often larger than the program itself. To further complicate matters, individual development documents usually only describe a few aspects of the program in question.

However development documents are often highly structural in nature. This structure results in explicit and implicit relationships both among development documents and between these documents and the program source documents. Understanding these relationships provides valuable insight into the functionality and structure of the program and issues that reflect why the program was implemented the way it was. Furthermore these relationships can be used to weave a 'literate program' that contains both text and graphics.

Unfortunately, the development documentation associated with large software projects is rarely indicative of the current state of the programs. Belady and Lehman [2] proposed three laws of program evolution, two of which indicate why it is important to spend time updating the development documentation so it is consistent with the current state of the program.

1. "Law of continuing change. A system that is used undergoes continuing change until it is judged more cost-effective to freeze and recreate it."
2. "Law of increasing entropy. The entropy of a system (its un-structured-ness) increases with time, unless specific work is executed to maintain or reduce it."

The first law indicates that maintenance is likely to be carried out in the future, and the second that this maintenance will be more difficult unless specific action is taken now. By making a program literate, its structure is captured and reinforced, easing the load on both current and particularly future activities that involve program comprehension.

3 Architecture of UQ✯

The UQ✯ system is based on a structured document model [30]. It allows for the representation of the syntactic and semantic structure of documents. It provides document construction facilities via textual and graphical views of document structures. Documents may also be constructed by integrated software-analytic tools and by importing text files.

The architecture providing this support is a persistent store of documents manipulated by *front-end* and *back-end* tools, as depicted in *figure 1*. *Front-end* tools are those which interact with users to present and modify the document store, while *back-end* tools are those which perform analytic operations upon the documents, producing results which become part of the document store.

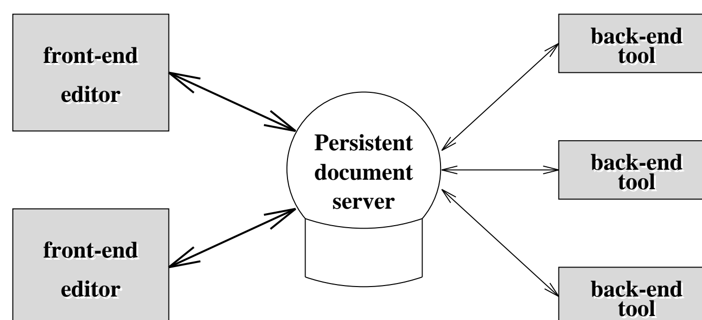

Figure 1. A high level view of UQ✯ architecture

3.1 Documents

The syntactic structure of traditionally textual documents such as Pascal programs can be expressed via an EBNF document which is in the document store. From this, the system produces a state machine used for scanning and a grammar for parsing. These are used by the document construction facilities in textual views to build parse tree documents from text input in the language specified by the EBNF grammar. These parse trees are the document representation; textual views are formed by unparsing.

The document store consists of atomic elements and relationships between these elements, such as a *Statement* parse tree node and its parent-child relationship to its syntactic constructs. Other document structures, such as the inter-relationship of the specification, design, implementation, and the user manual of a software system can also be expressed.

3.2 Relations

Relations exist between document segments. Relations can be classified by the mechanism by which they are created. The categories of relations supported by UQ✯ are:

- User-Defined Values. The user creates individual links between document components. For example, a user might link portions of the implementation of an algorithm to points in a textual discussion of the algorithm in the design documentation.
- Pre-defined. The UQ✯ parsing mechanisms create hierarchic links in parsed documents. Also, EBNF grammars are presented as relations among syntactic constructs.
- User-Defined Derived. The user expresses a relation in terms of other relations. For example, a user might define a call-graph from relations defined by tools and the parsing system. This example is illustrated in section 4.
- Tool-Defined. A tool defines new relations based on existing relations. For example, a back-end tool (*figure 1)* might generate a relation describing the scope of variables from the syntactic structural relations.

3.3 Back-end Tools

Back-end tools can be used to contribute to the document store. They may generate relations between document components, and generate document components for either inspection or further manipulation by the user. The nature and contribution of such tools are varied. A compiler may produce a list of errors, or an executable. A constraint verification tool may produce a single yes/no value based on its analysis. A program dependency analysis tool may produce a set of dependency relations. These are only a few examples of back-end tools that may be coupled with UQ✯. A simple example of a possible application of a back-end tool is highlighted in the example presented in section 4.

3.4 Front-end Tools

The user interacts with the underlying documents via *views* of the structure provided by front-end tools. For interactive presentation and manipulation, the view can be some combination of text and graphics ranging from an automatically formatted unparsing to a graph of nodes and arcs. Similar view generation is able to generate publication-quality presentations, by generating LaTeX input.

4 A Simple Program Dependency Example

In this section we present an example to highlight the flexibility of definition and use of relations with UQ✯. The example is a user-derived relation for a call graph in Modula-2. A call graph is a directed graph depicting the procedure call structure of a program. It is defined by the relation between each procedure or module in a program and each procedure it calls.

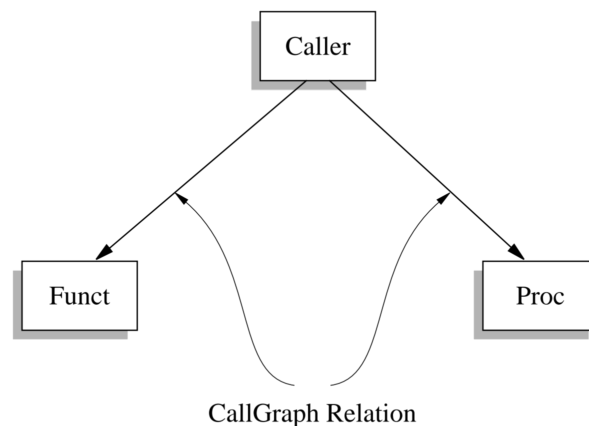

Figure 2. Call graph showing derived *CallGraph* relation

The example is illustrated using the Z notation [25] to show how a useful relation can be derived from other given relations. The relation described is the *CallGraph* relation (*figure 2*). *Figure 3*, at the end of the section, provides an overview of the example under discussion.

We start by defining the set of all possible nodes in the parse tree as:

[NODE]

A relation is a segment that relates two other segments. We therefore define a relation as a mapping between two NODE elements. For this example, three relations are pre-defined:

$GrammarNodeType : NODE \leftrightarrow NODE$
$Parent : NODE \rightarrow NODE$
$DeclarationUse : NODE \leftrightarrow NODE$

The first two relations must exist for all documents. *GrammarNodeType* is a relation between a parse-tree node and its grammar-node and is inherent in the EBNF of the document. *Parent* is a relation between a child node and its parent node and is inherent in the structure of the parse tree. In this example *Parent* is defined as a partial function which is a specialised relation. *DeclarationUse* is a relation between the declaration of a program element and all uses of that element and is assumed to have been constructed by a back-end tool familiar with the semantics of Modula-2.

The *DeclarationUse* relation contains some information that is not relevant to call graphs since call graphs only consider procedure declarations, not all program elements. To extract the required information, a derived relation *ProcDecProcUse* is defined. This relation contains only those (declaration, use_id) pairs where the declaration is a procedure declaration and the parent node in the parse tree of the use_id is a procedure or function call.

$$
\begin{array}{l}
ProcDecProcUse : NODE \leftrightarrow NODE \\
\hline
\forall\ declaration, use_id : NODE \bullet (declaration, use_id) \in ProcDecProcUse \Leftrightarrow \\
\quad (declaration, use_id) \in DeclarationUse \land \\
\quad (declaration, \text{“ProcedureDecl”}) \in GrammarNodeType \land \\
\quad (\quad (Parent\ (use_id), \text{“ProcedureCall”}) \in GrammarNodeType \lor \\
\quad\quad (Parent\ (use_id), \text{“FunctionCall”}) \in GrammarNodeType\ )
\end{array}
$$

Two further derived relations are required to define the *CallGraph* relation: *ModOrProcParent* and *NearestModOrProc*. *ModOrProcParent* relates each node to every enclosing procedure or module declaration.

$$
\begin{array}{l}
ModOrProcParent : NODE \leftrightarrow NODE \\
\hline
\forall\ descendant, ancestor : NODE \bullet (descendant, ancestor) \in ModOrProcParent \Leftrightarrow \\
\quad (descendant, ancestor) \in Parent^{+} \land \\
\quad (\quad (ancestor, \text{“ProcedureDecl”}) \in GrammarNodeType \lor \\
\quad\quad (ancestor, \text{“ModuleDecl”}) \in GrammarNodeType\ )
\end{array}
$$

NearestModOrProc is an additional specialisation that relates each node to the closest enclosing procedure or module declaration.

$$
\begin{array}{l}
NearestModOrProc : NODE \leftrightarrow NODE \\
\hline
\forall\ descendant, nearest : NODE \bullet (descendant, nearest) \in NearestModOrProc \Leftrightarrow \\
\quad (descendant, nearest) \in ModOrProcParent \land \\
\quad \neg\ (\ \exists\ another : NODE \bullet another \neq nearest \land \\
\quad\quad (descendant, another) \in ModOrProcParent \land \\
\quad\quad (another, nearest) \in ModOrProcParent\ )
\end{array}
$$

From these derived relations, the *CallGraph* relation can be defined. A callee is any procedure declaration in *ProcDecProcUse* that has a corresponding caller in *NearestModOrProc*.

$$
\begin{array}{l}
CallGraph : NODE \leftrightarrow NODE \\
\hline
\forall\ caller, callee: NODE \bullet (caller, callee) \in CallGraph \Leftrightarrow \\
\quad \exists\ use_id : NODE \bullet \\
\quad\quad (callee, use_id) \in ProcDecProcUse \land \\
\quad\quad (use_id, caller) \in NearestModOrProc
\end{array}
$$

As with the other relations derived in this example, the *CallGraph* relation may be used as a starting point for deriving other useful program dependencies.

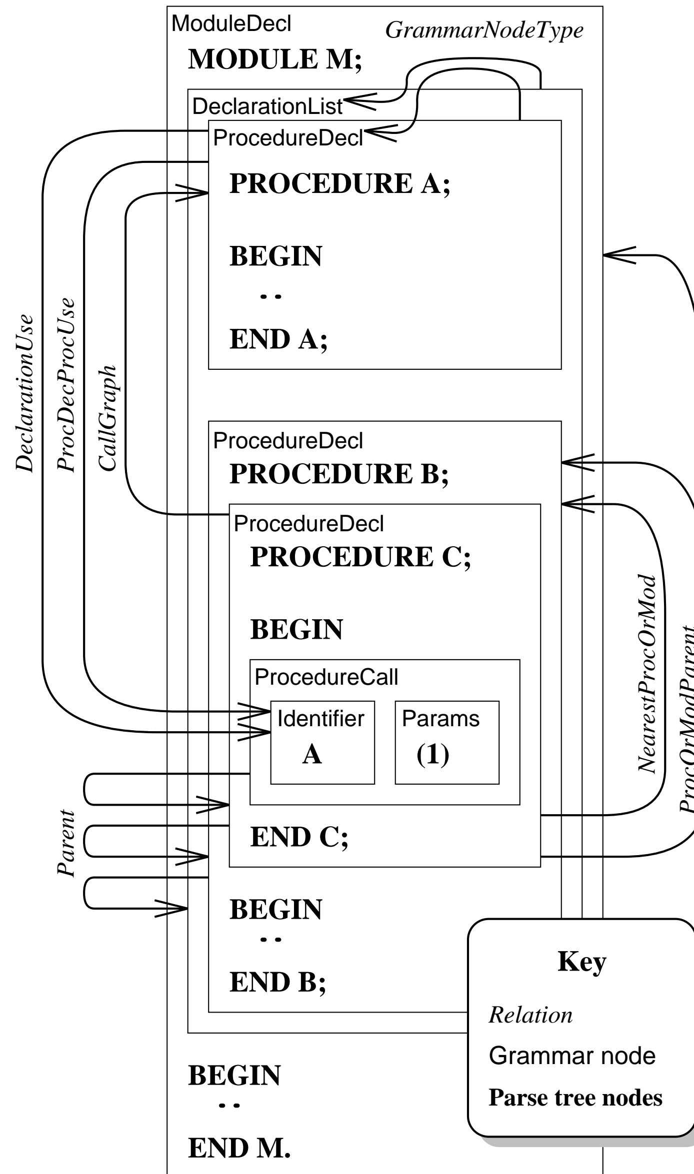

Figure 3. Relations involved in the derivation of the CallGraph relation

5 Presentation and Navigation through Documents

In this section we illustrate how relations can be used by front-end tools to allow the programmer to navigate through a collection of documents. Our illustration traces a enquiry session of a simple program that starts with a call graph and allows the programmer to browse the program progressively in more detail.

Figure 4 shows a call graph for a program that calculates the greatest common denominator for a set of positive integer pairs. Such a graphical view may allow the user to select a node and view the related code. The *CallGraph* relation was described in the previous section as a relation between the caller and the callee, which represented the arcs in the call graph. The nodes in the graph are the ProcedureDecl nodes in the parse tree (*figure 3*). Thus by selecting a node in the call graph, UQ✭ is given a handle to the corresponding node in the parse tree.

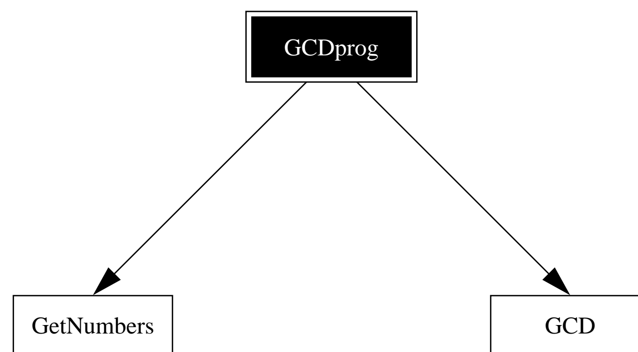

Figure 4. Call Graph of GCDprog

In *figure 4* the procedure GCDprog has been selected. GCDprog is the root module with the same name. *Figure 5* shows GCDprog as well as text and Z documents that describe it. A relationship exists between the GCDprog module and the text and Z documents. This relationship allows UQ✭ to establish this view.

Additionally the detail of the procedure declarations in GCDprog is suppressed. The use of detail suppression [4, 29] supports program comprehension by reducing the amount of information the programmer has to contend with. However the programmer may at some stage be interested in these procedures.

View generation allows the programmer to expand the procedure declaration in situ, but this would make presentation of associated documentation more cluttered. Although the programmer may be interested in one of the two procedures, the information contained in *figure 5* may also be required. Opening another window to display a chosen procedure addresses both of these issues. An example of such a window for GCD is shown in *figure 6*.

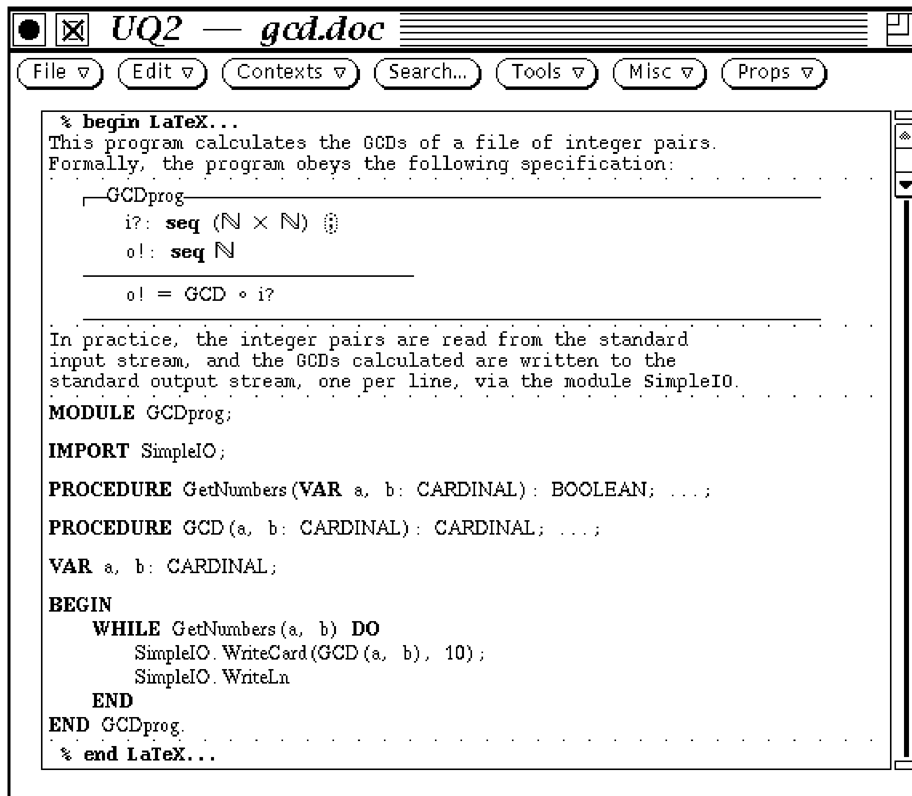

Figure 5. View if GCDprog and associated description documents

Oman [20] emphasises that off-line presentation of program sources are still a necessity even with the availability of multiple window system like UQ✫, and proposes a book paradigm for the presentation of such documents. UQ✫ is ideally suited to producing such documents through its ability to interleave multiple document types, include LaTeX commands in text documents and generate views from cross-reference (relation) information.

For the example described in this section, a presentation promoted by the book paradigm may have the following contents page:

Contents

Structure chart 1
GCDprog 2
GCD 3
GetNumbers 4
Index 5

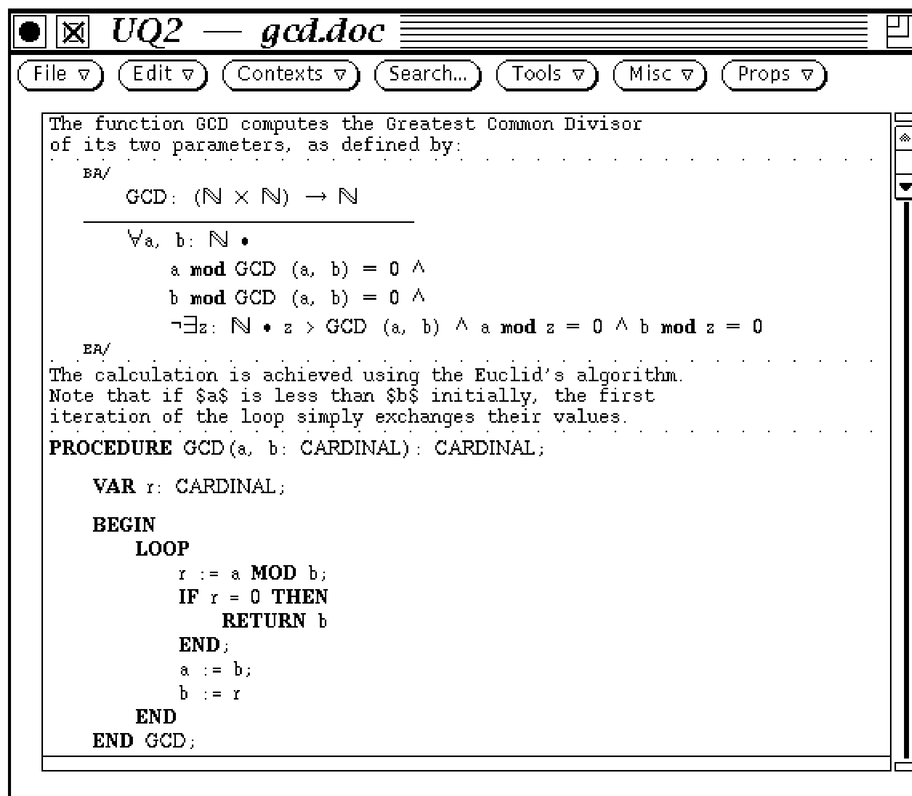

Figure 6. View of GCD and associated description documents

The structure chart is the view presented in *Figure 4*. The GCDprog and GCD would be the views presented in *figures 5* and *6* respectively and GetNumbers would be the equivalent view for that procedure. The index would contain page and possibly line numbers for the declaration and the uses of program elements occurring in the program. Note that this can be constructed from the *DeclarationUse* relation presented in the example in section 4.

6 Conclusion

Program comprehension is an important issue in software engineering. This paper has identified two techniques that can be employed, both to comprehend existing programs and to encourage the development of more comprehensible programs. They are, analysis of program dependencies and literate programming.

Documents that describe a software system are often highly structural in nature. This structure results in explicit and implicit, inter- and intra-document relationships. These relationships may be used in program dependency analysis or to link parts of various documents together to form literate programs.

We described UQ✯, an integrated development environment which supports the use of multiple documents and document types and allows the relationships that are implicit in a set of documents to be represented explicitly. It will allow existing programs and related documentation to be imported for analysis and construction into a literate program. Furthermore it encourages the literate development of new programs, by providing the basis for hyper-textual literate views of programs. We illustrated a possible definition and use of relations within the UQ✯ environment in two short examples. The first presented the derivation of a simple derived dependency and the second illustrated the use of relations for navigation through and presentation of multiple documents of differing descriptions. The relations and views presented in these examples are only a limited set of the views that UQ✯ will be capable of presenting.

To conclude, we believe that an environment that treats relations as a fundamental part of document structure is both flexible in definition and nature.

7 Acknowledgements

The work described in this paper is an extension of work coordinated by Professor Jim Welsh and performed at the University of Queensland over recent years. The UQ✯ environment is based on work done on UQ1 [31] and UQ2 [6, 29]. Early investigations of document based processing was performed by Jun Han [30] and contributed to the relational approach adopted in this paper. Current research is investigating coherent mechanisms for the graphical presentation and manipulation of software documents in the UQ✯ environment. Much of this work, including the current research has received funding from the Australian Research Council.

We would specifically like to acknowledge Professor Jim Welsh for proposing the paper, Kelvin Ross for contributions to the definition of the Z schemas used in the example and Tim Mansfield for reviewing the paper.

References

[1] Ambras J. P., Berlin L. M., Chiarelli M. L., Foster A. L., O'Day V., Splitter R. N. Microscope: An integrated program analysis toolset. *Hewlett-Packard Journal*, 39(8):71–83, August 1988.

[2] Belady L., Lehman M. A model of large program development. *IBM Systems Journal*, 3:225–52, 1976.

[3] Brade K., Guzidial M., Steckel M. Soloway E. Whorf: A hypertext tool for software maintenance. *International Journal of Software Engineering and Knowledge Engineering*, 4(1):1–16, January 1994.

[4] Broom B., Welsh J. Detail suppression systems for interactive program display. In *Proceedings of the 9th Australian Computer Science Conference*, pages 83–93, January 1986.

[5] Broom B., Welsh J. Another approach to literate programming. In *Proceedings of the 11th Australian Computer Science Conference*, pages 257–68, Brisbane, February 1988.

[6] Broom B., Welsh J., Wildman L. UQ2: A multilingual document editor. In *Proceeding of the 5th Australian Software Engineering Conference (ASWEC '90)*, pages 289–94, Sydney, May 1990.

[7] Brown G. P., Carling R. T., Herot C. F., Kramlich D. A., Souza P. Program visualization: Graphical support for software development. *Computer*, 18(8):27–35, August 1985.

[8] Chen Y. F., Grass J. E. The C++ Information Abstractor. In *Proceedings of the Second C++ Conference*, pages 34–50. USENIX, April 1990.

[9] Chen Y. F., Ramamoorthy C. V. The C Information Abstractor. In *Proceedings of the Tenth International Computer Software and Application Conference*, pages 291–98. COMPAC, IEEE Computer Society Press, October 1986.

[10] Chen Y.F., Nishimoto M. Y., Ramamoorthy C. V. The C Information Abstraction system. *IEEE Transactions on Software Engineering*, 16(3):325–34, March 1990.

[11] Cleveland L. A user interface for an environment to support program understanding. In *Proceedings of the '88 Conference on Software Maintenance*, pages 86–91, Phoenix, Arizona, October 1988. IEEE Computer Society.

[12] Cleveland L. A program understanding support environment. *IBM Systems Journal*, 28(2):324–44, 1989.

[13] Corbi T. A. Program understanding: Challenge for the 1990s. *IBM Systems Journal*, 28(2):294–306, 1989.

[14] Harrold M. J., Soffa M. L. Computation of interprocedural definition and use dependencies. In *Proceedings of the 1990 IEEE International Conference on Computer Languages*, pages 297–306. IEEE Computer Society, 1990.

[15] Keables J., Roberson K., Mayrhauser A. Data flow analysis and its application to software maintenance. In *Proceedings of the '88 Conference on Software Maintenance*, pages 335–47, Phoenix, Arizona, October 1988. IEEE Computer Society.

[16] Knuth D. E. Literate programming. *The Computer Journal*, 27(2):97–111, February 1984.

[17] Mayrhauser A. von, Vans A. M. From code understanding needs to reverse engineering tool capabilities. In *Proceedings: International Conference on CAiSE '93*, pages 230–39. IEEE Computer Society, 1993.

[18] Meekel J., Viala M. LOGISCOPE: A tool for maintenance. In *Proceedings of the '88 Conference on Software Maintenance*, pages 328–34, Phoenix, Arizona, October 1988. IEEE Computer Society.

[19] Oman P. Maintenance tools. *IEEE Software*, 7(3):59–65, May 1990.

[20] Oman P., Cook C. The book paradigm for improved maintenance. *IEEE Software*, 7(1):39–45, January 1990.

[21] Podgurski A., Clarke L. A. A formal model of program dependencies and its implications for software testing, debugging and maintenance. *IEEE Transactions on Software Engineering*, 16(9):965–79, September 1990.

[22] Reiss S.P. Pecan: Program development systems that support multiple views. *IEEE Transactions on Software Engineering*, SE-11(3):276–84, March 1985.

[23] Richardson D. J., O'Malley T. O., Moore C. T., Aha S. L. Developing and integrating ProDAG in the Arcadia environment. In Herbert Weber, editor, *Proceedings of the Fifth ACM SIGSOFT Symposium on Software Development Environments*, volume 17 of *Software Engineering Notes*, pages 109–119, Tyson's Corner, Virginia, USA, December 1992. Special Interest Group on Software Engineering, ACM Press.

[24] Schwanke R. W., Platoff M. A. Cross references are features. In *ACM 2nd International Workshop on Software Configuration Management*, pages 86–95, Princeton, N. J., October 1989. ACM Press.

[25] Spivey J. M. *The Z notation: A reference manual*. Prentice Hall International, 1989.

[26] Teitelman W. A display oriented programmer's assistant. *International Journal of Man-Machine Studies*, 11:157–87, 1979.

[27] Weiser M. Program slicing. In *Proceedings of the 5th International Conference on Software Engineering*, pages 439–49, San Diego, California, March 1981. IEEE Computer Society.

[28] Weiser M. Programmers use slices when debugging. *Communications of the ACM*, 25(7):446–52, July 1982.

[29] Welsh J., Broom B., Kiong D. A design rationale for a language-based editor. *Software–Practice and Experience*, 21(9):923–48, 1991.

[30] Welsh J., Han J. Software documents: Concepts and tools. Technical Report TR-93-23, University of Queensland, Brisbane, Australia, 1993.

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

[32] Wilde N. Understanding program dependencies. Curriculum Module SEI-CM-26, Camegie Mellon University, August 1990.

[33] Wilde N., Huitt R., Huitt S. Dependency analysis tools: Reusable components for software maintenance. In *Proceedings of the '89 Conference on Software Maintenance*, pages 126–31, Miami, Florida, October 1989. IEEE Computer Society.

[34] Yau S. S. Methodology for software maintenance. Technical Report RADT-TR-83-262, Rome Air Development Centre, Griffis Air Force Base N. Y., February 1984.

[35] Yau S. S., Collofello J. S., MacGregor T. Ripple effect analysis of software maintenance. In *Proceedings of COMPSAC '78*, pages 60–65. IEEE Computer Society, 1978.