SOFTWARE VERIFICATION RESEARCH CENTRE

DEPARTMENT OF COMPUTER SCIENCE

THE UNIVERSITY OF QUEENSLAND

Queensland 4072
Australia

TECHNICAL REPORT

No. 93-6

Deriving Modular Designs from Formal Specifications

David Carrington and David Duke and Ian Hayes and Jim Welsh

April 1993

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

Deriving Modular Designs from Formal Specifications

David Carrington* and David Duke† and Ian Hayes* and Jim Welsh*

September 30, 1993

Abstract

We consider the problem of designing the top-level modular structure of an implementation. Our starting point is a formal specification of the system. Our approach is to analyse the references to the state variables by the operations of the system. Those variables that are referenced/modified together are likely candidates for forming the state of a module. We evaluate the strategy by applying it to a large Z specification of a language-based editor.

1 Introduction

We would like to develop techniques that assist software engineers to modularise a system. Our starting point for system development is a formal specification consisting of a state, typically containing a large number of variables, and a set of operations on the state. We consider the problem of developing modular structures suitable for an implementation starting from the specification, but not (necessarily) using the explicit structure of the specification.

In Section 2, we explain our approach as a two-step process:

- analysing the formal specification to produce cross-reference information between the specification state variables and the top-level operations,
- synthesizing a collection of modules from this cross-reference.

A further projection stage (discussed briefly in section 2.3) is required to define the signatures and pre- and post-conditions of the module-level operations.

As a case study, in Section 3 we examine the application of the techniques to a Z [8, 16] specification of a language-based editor [18, 1]. In the remainder of the introduction, we discuss the multiple roles of formal specifications and the concept of modularity.

1.1 Formal specifications

Formal specifications are gaining acceptance as an important component of methods for developing high-quality software. A formal specification has many roles in the development process. Some examples are:

*Department of Computer Science, The University of Queensland, Brisbane, 4072, Australia

†Department of Computer Science, University of York, Heslington, York, YO1 5DD, U.K.

- as a vehicle for precise communication between the customer and the software supplier,
- as the standard against which the final program code is verified, and
- as a model from which it is possible to prove properties as an aid to validating the formal specification against the informal requirements.

In this paper, our interest is on using formal specifications as a starting point for the software development process. While this might seem the most obvious use of a formal specification, people such as Hall [6] claim it is not the most important and that formal specifications are valuable without using them in a formal development process. The latter claim is undoubtedly true. However, without a direct link between the formal specification and the software development process, the task of verifying that the program code satisfies the specification rapidly becomes infeasible with increasing system size.

We are interested in techniques that start with a formal specification and systematically incorporate design decisions leading to the program code. Such techniques provide an audit trail of the design process that can be checked independently. This is invaluable for future maintenance.

Existing formal development techniques such as VDM [9] and the refinement calculus [11] provide techniques such as data reification (refinement) but there is no guidance in these methods for achieving suitable modularity.

1.2 Modularity

An often-stated objective in software development [15, page 68] is to achieve a design that is easy to understand and to modify. In a large system, one should be able to understand each component independently, rather than having to examine each in the context of the rest of the system. The ideal is to have a design in which changes to the system can be isolated to changes in a single component. The choice of components and their composition is significant in determining the overall complexity of the system and so contributes to the ease with which the system can be understood. It is also significant for the development process since

- programming and testing of each component can be performed independently, and
- parts of the system can be changed throughout the system's lifetime without necessarily revising or retesting other parts.

To achieve these goals, the choice of structure is influenced by factors other than the system's required functionality.

Modularisation aims to manage the complexity of software — an example of the divide and conquer strategy. Meyer [10, page 11] claims that modularity is one of the favourite buzzwords of software engineering. Certainly modularity is a concept that is widely promoted but it is not always defined consistently. Yourdon and Constantine [21, page 32] state:

> A module is a lexically contiguous sequence of program statements, bounded by boundary elements, having an aggregate identifier.

This very general definition encompasses procedures, functions, coroutines, and tasks and relates primarily to the functional-design methods where single-entry single-exit modules are emphasised.

Parnas' widely-quoted 1972 paper [13] established the *information-hiding* principle as a basis for modularisation. The principle states that every module should be characterised by its knowledge of a design decision that it hides from all others. As a consequence, data structures and their accessing and modifying procedures are typically part of a single module. This view of a module did not suit the syntactic structures of programming languages of that time and it has taken considerable time to become accepted. Abstract data types, programming languages such as Modula-2 and Ada, and growing interest in object orientation have all served to make Parnas' view of modules more widely accepted.

The information-hiding principle is not sufficient to determine a unique module structure. Other qualitative criteria that are used to guide modular decomposition are *cohesion* and *coupling* [21]. These criteria were originally proposed for functional-design modules but the definitions can be extended for information-hiding modules [5, 14].

The aim of a modular decomposition is to produce a *loosely-coupled* set of modules, each of which is *highly cohesive.* A cohesive module is responsible for a single part of the functionality of the system. In the context of a state-based specification of a module, one way of interpreting this is to consider how the operations act upon the state. For example, if a subset of the operations require only a subset of the state to define their effect, then one can argue that the module is not cohesive and should be split. On the other hand, if operations of one module need access to the state variables of another module in their definition, the two modules are tightly coupled, and should perhaps be merged or regrouped. The two criteria are not independent and the design challenge is to achieve a decomposition that is satisfies both criteria. It is easy to improve one criterion at the expense of the other, for example splitting a module to improve cohesion but introducing additional coupling.

2 Modular design method

The simplest approach to structuring a design is to use the structure of the specification. This approach however can be unduly restrictive. As explained in Section 1, the specification has many roles in the software development process and its structure must to chosen to satisfy the combination of these roles. For this reason, we take a less restricted approach that does not use the explicit specification structure directly, rather it relies on our analysis of the specification as a guide in determining a suitable modular structure. Of course, it is still possible that parts of the implementation structure could follow that of the specification. In that situation, the job of synthesizing the corresponding module specifications is simplified.

Before we delve into the details of techniques for modularising large systems, it is appropriate to point out that the first step in the refinement of such a system may not be devising a modular structure, but may be a top-level data refinement that changes the representation of the system state. However, once this data refinement has been performed, modularisation of the system may be the appropriate next step. In general, we see the modularisation techniques outlined below as being applicable at any stage of the refinement process, but we have concentrated on applying them to the top-level specification because we see that as a likely place for the techniques to be exercised.

In Section 2.1, we describe a method for analysing the patterns of reference by operations to state variables with the aim of grouping variables that are referenced together to form the state of a module. We also consider what additional specification-level information or structure can assist in this process (without compromising the specification by overloading it with design information). In Section 2.2, we consider how to use the information derived from analysing the specification to synthesise a suitable modular structure by partitioning the state space.

One advantage of the two-step process is that only the first step is specific to a particular specification notation and our approach should be adaptable to other model-based specification languages, such as VDM.

In order to achieve the goal of a loosely-coupled set of cohesive modules in the implementation, we argue that the first step should be to partition the abstract specification into a loosely-coupled set of cohesive abstract modules. We do this by partitioning the abstract state of the system and taking projections of the system operations. If we partition the abstract state in a manner that leads to a cohesive set of modules acting on the abstract state, the concrete implementation should follow suit. To put it another way, if the abstract views of the modules are highly coupled or not cohesive, it is likely that the concrete ones will be as well. The partitioning of the state and operations into modules can be viewed as a combination of data refinement and procedural refinement.

2.1 Analysing the specification

Given a specification that describes a system in terms of a state and operations on that state, the goal of analysis is to produce a table cross-referencing state variables and operations. Each entry in the table has one of three values:

eq (equated) the operation neither references nor changes the value of the variable;

rd (read) the operation references the variable but leaves its value unchanged;

wr (written) the operation potentially changes the value of the variable (and may also reference its value).

Having extracted this information, we can group operations according to the variables they refer to and/or modify. A useful extension is to consider grouping operations according to the *sets* of variables that they refer to and/or modify.

We should include a warning here: just because variables are always referenced together does not mean that they should be part of the same module. They may have quite different purposes that could be easily split into separate modules. The converse is, however, more likely: if variables are not referenced together, then it is unlikely that they will form a suitable grouping for the state of a module.

Identifying variables explicitly as being completely derived from a set of other variables may also be useful, because if none of the other variables are modified then neither is the derived variable. There is not necessarily a unique way of classifying variables as derived or non-derived. For example, a variable p may be derivable from a set of other variables, $p = f(q, r)$. Depending on the function f, it may be possible to express the relationship as $q = g(p, r)$. In addition, p may be also derived from a different set of variables e.g. $p = h(s, t)$. To record the derived relationship between variables, we need to record for each variable, minimal sets of other variables from which it can be derived.

In the Z specification of the UQ2 editor, for example, the state of the edited document is represented by the string of characters preceding the cursor, *upstream*, and the string following the cursor, *downstream*. As well, there is a derived variable *str*: the complete document string, that is, the concatenation of *upstream* and *downstream*. This state is described in Z by declaring all three variables and linking them with a coupling invariant.

$$\begin{array}{|l}
\multicolumn{1}{l}{\mathit{DOC}} \\
\hline
\mathit{upstream}, \mathit{downstream}, \mathit{str} : \mathrm{seq}\ \mathit{Char} \\
\hline
\mathit{str} = \mathit{upstream} \frown \mathit{downstream} \\
\hline
\end{array}$$

Although we have described *str* as a derived variable, for this example, the value of any of these three variables can be derived from the values of the other two. However this fact is based on the semantic properties of the '$\frown$' operator and can not be deduced syntactically. Such closely coupled variables complicate the analysis a little as we must be careful to examine the effects of the coupling invariant. In practice, however, such closely coupled variables do not cause a significant problem for modularisation, as they are typically grouped into the same module by the partitioning process.

Other than ensuring that we recognise the coupling between the variables in the analysis phase, we do not have to treat derived variables specially. Once the variables are all in a single module, it is likely that they will be data refined as a group. For example, the editor document state *DOC* may be represented by the complete document string plus an index representing the current cursor position.

2.1.1 Analysing Z specifications

In this section, we describe two specific examples of issues that arise with the Z specification notation:

- The open-world view of Z makes it difficult to determine which state variables are potentially changed by an operation. The practice of using additional state variables to simplify the expression of the specification but whose value is determined by other variables also complicates the analysis process.

- Z specifications are often constructed in a *bottom-up* style so that the reader is introduced gradually to the concepts in the system. Examples are [8, the block-structured symbol table chapter in Part A] and [12]. The technique commonly used to achieve this is called framing or promotion. Naive analysis of promoted operations fails to achieve a sufficiently fine-grained result so special techniques have been developed.

For a specification written in a notation such as Z that uses the predicate calculus, establishing that a variable is unchanged by an operation would, in general, require theorem proving. In practice, however, it is desirable to automate the analysis of specifications, and consequently we have settled for a syntax-based approximation to the result implied by a semantic analysis. By approximation we mean that our analysis may indicate that a variable is referenced (**rd** or **wr**) by an operation although it could be shown from the semantics of the specification that the variable is unchanged or need not be referenced. This apparently 'pessimistic' view is founded on the semantics of Z, and in particular, the interpretation given to operation specification. An operation is defined by relating the value of state variables before the operation to their value after. Unless otherwise constrained, a variable in a Z specification is free to take any value within its type. Consequently, unless the before and after values of a variable are constrained to be equal (the variable is said to be *equated*), the value of that variable may be changed by the operation. At the level of modular design, this corresponds to the operation having write access to the variable.

This issue raises the question of what additional information can be supplied with the specification to aid in analysing the specification, without compromising its role.

Other specification methods provide more help in this area. For example, VDM [9] uses *read* and *write* imports to identify explicitly state variables that are referenced without modification and with the possibility of modification, respectively. In Object Z, *delta* lists provide the state variables that may be changed [2, 4].

The promotion technique allows operations specified on state S to be promoted to operations on a larger state that contains S as a component. A typical example is the promotion of an operation on a state S:

$$
\begin{array}{|l}
\multicolumn{1}{l}{S} \\
\hline
a : A \\
b : B \\
c : C \\
\hline
P(a, b, c) \\
\hline
\end{array}
$$

to operate on a state of the form:

$$
\begin{array}{|l}
F : X \pfun S
\end{array}
$$

so that the operation takes place on just one element of the function F.

A naive approach to the analysis of such a specification would treat F as a single state variable and, if a promoted operation modifies any of the components of S, it would be considered to modify F.

To get a more detailed view of variable access in our analysis, there is a simple transformation that can be applied to the specification so that the components a, b and c are treated separately. First we replace F by,

$$
\begin{array}{|l}
Fa : X \pfun A \\
Fb : X \pfun B \\
Fc : X \pfun C \\
\hline
\mathrm{dom}\, Fa = \mathrm{dom}\, Fb = \mathrm{dom}\, Fc \\
\forall x : \mathrm{dom}\, Fa \bullet P(Fa(x), Fb(x), Fc(x)).
\end{array}
$$

The relationship between this new state and F is,

$$
\begin{array}{l}
Fa = (\lambda\, x : \mathrm{dom}\, F \bullet F(x).a) \\
Fb = (\lambda\, x : \mathrm{dom}\, F \bullet F(x).b) \\
Fc = (\lambda\, x : \mathrm{dom}\, F \bullet F(x).c)
\end{array}
$$

In the specification any references to $F(x).a$ can be replaced by $Fa(x)$, etc.

With this separated state, it is possible to achieve a finer granularity of analysis on the state variables of the system.

Note that the form of the above description is intended only to give a theoretical view of how a finer granularity can be achieved. In practice, an analyser may not actually transform the specification before doing the analysis. It may directly analyse the specification to come up with the same results. For example, it may indicate that $F(*).a$ is modified to indicate that the transformed Fa is modified.

2.2 Synthesizing modules

With the information that can be gathered about groupings of operations with respect to groupings of state variables, it is possible to make reasonable attempts to partition the state variables into modules. The analysis gives a guide to how to partition the abstract state, but the final decision is left to the judgement of the designer who may take other factors into account. The aim is to choose a modular structure that minimises the coupling required between modules and maximises the cohesion within a module. The designer who may be influenced by additional information not available to the analysis process, such as the availability of existing modules. The role of the analysis and synthesis results is to provide the designer with a global view that can act as the first draft of the top-level modularisation. With a large specification, it is easy to forget about some of the interconnections if the modularisation is performed manually. Without domain knowledge however, the automatic process is inevitably incomplete and will require fine tuning by the designer.

An initial partitioning can be formed by grouping together sets of operations that modify and/or reference similar sets of state variables. This can be further refined by recognising that some sets in the partition can be further subdivided because, although they reference similar variables, they perform logically distinct functions. This second step requires domain knowledge and hence is not automatable.

The initial partitioning is not necessarily simple and techniques to deal with problems that arise must be available. One common situation is that a particular state variable is referenced from two distinct sets of operations. A simple example is given in Figure 1 where the variable b is referenced by both operations (a blank entry denoted the *equated* relationship).

	Variables		
	a	b	c
Operation-1	w	r	
Operation-2		w	r

Figure 1: Example of overlapping state spaces

Of course, there may be more than just a single variable in the overlap between the two distinct sets of operations, and there may be three (or more) distinct groups with variables in common to all groups. Below we concentrate on the special case of a single variable that is in the overlap between two modules, as this allows a simpler presentation of the proposed strategies. The techniques for handling the more general case are straightforward extensions of the strategies discussed below.

When a state variable is referenced from two distinct sets of operations, it makes sense to structure the system into two modules with the overlap between the state spaces either contained in a third shared sub-module or duplicated in both modules.

Shared sub-module With this technique, the structure chosen consists of two top-level modules with a shared subordinate module. The strategy of hiving off common parts into a separate, subordinate module will be recognised as a basic technique used in engineering software. The analysis of the specification can aid in identifying suitable sub-module states. This is beginning to create a hierarchy of modules that will be extended as the design process continues. For the example in Figure 1, the variable b would be the state of the subordinate module.

Duplicated variables Another possibility in resolving the partitioning problem is to allow some state variables to be duplicated in different modules. If operations can nondeterministically choose the value of the duplicated variable, care needs to be taken to ensure that the choice is made consistently in the two modules. Effectively, there is a global invariant in the specification equating the duplicate variables. If there is a choice for the variable value then we require that the two modules make the same choice. In practice, we would like to avoid such global invariants (a form of module coupling) and hence would only duplicate variables that are uniquely determined by all operations. If there are multiple choices for the value of a variable, then it is simpler to make the choice unique by a top-level refinement before duplicating the variable and partitioning the state.

Designers will be familiar with the strategy of duplicating information — usually in different representations — to provide efficient implementations for different groups of operations of a system. This strategy can be incorporated into our modularisation method during the partitioning of the state, although perhaps we should point out that we use the word 'partition' loosely, as the sets in our partition are not necessarily disjoint with this strategy.

2.2.1 Synthesis Techniques

The aim of the synthesis process is to transform the specification (as represented by the cross-reference table) into an equivalent collection of modules where each module captures some of the functionality of the original system. Thus each module can be represented by its own cross-reference table.

To direct this transformation, we are developing quantitative measures for cohesion and coupling in terms of the cross-reference table values. A possible definition for the cohesion of a single module is:

$$\text{cohesion}(\text{M}) = \frac{\text{no. of read or written entries}}{\text{total no. of entries in } M\text{'s cross-reference table}}$$

with the overall cohesion of a design defined as the minimum cohesion over all modules. Our view of coupling is simple: two modules are coupled if they share a common variable or if an operation requires access to them both. The actual number of shared variables is not considered important. A definition for the coupling of a design is the number of coupled modules divided by the potential number of coupled modules. Both measures lie in the range 0 to 1.

Various clustering and grouping algorithms are being investigated to help identify potential modules. The intention is to highlight patterns of reference to subsets of variables within the system state by subsets of operations. Such subsets of the state variables are candidates for the state of a cohesive module.

Splitting The splitting approach treats the original specification as a single module with no coupling but low cohesion. This module is then partitioned into two or more modules, each with higher cohesion than the original. The splitting process can then be repeated for each module whose measure of cohesion is still considered unsatisfactory.

Joining This approach starts with every variable in a separate module and selects a pair of highly-coupled modules to combine into a new composite module. The pair is chosen so as to minimise the coupling of the new design. The coupling arises from operations that require access to multiple modules. The joining process is continued until the level of coupling between any two pairs of modules is considered appropriate.

Graph algorithms A further approach treats the cross-reference table as a graph with a *read* or *written* entry represented by an edge. The graph is then processed to extract a tree representing the module hierarchy [3]. The Choi and Scacchi algorithm was developed for extracting design descriptions from implementations. A tree structure representing the system's architectural design is derived from a network of relationships between implementation-level modules. The algorithm breaks the network into its biconnected components with the articulation points generating interior nodes of the tree.

For our synthesis phase, we can apply it to the cross-reference network to identify variables that are referenced by two distinct sets of operations. Another technique represents just the variables as nodes with an edge between two variables when the number of operations accessing those variables exceeds some threshold. A variety of designs can be generated by adjusting the threshold.

2.3 Designing module operations

Having identified the state variables that are to be grouped together to form the state of a module, we need to identify operations on this state. Here are two approaches:

- *posit* the operations on the module state and then implement the top-level operations in terms of these – the traditional approach, and
- *calculate* the projection of the top-level operations that refer to the state variables of the module – the calculus approach.

A combination of these approaches is likely to be more practical. The calculus approach can provide a first-level approximation to the required operations. Then a modicum of generalisation/abstraction/good taste can be applied to give a more widely applicable module design. Looking for commonalities amongst the projected operations is an obvious starting point for this process. An issue here is allowing for changes to the top-level specification, without necessitating changes to the module interfaces, only the way they are used. We want each module to provide a set of primitive operations from which more complex operations can be built, rather than a set of operations designed specifically for the current version of the application. For each top-level operation, it is then a matter of developing an implementation in terms of the operations provided by the modules.

3 UQ2 case study

To aid our evaluation of analysis and synthesis techniques, a substantial case-study problem in modularisation has been used. The case study focuses on a generic language-based editor (UQ2) under development at the University of Queensland. UQ2 is a multi-lingual editor designed to manipulate documents containing a variety of symbolic notations or languages in a closely interleaved fashion. Such documents arise in software development where specifications, refinements, proofs, programs and informal descriptions are generated.

The editor's facilities have been specified in Z. The specification is a substantial document of about one hundred pages which is structured to describe the editor in six stages. At each stage, additional elements of the editor state and relevant operations are introduced. Each stage builds on the previous stage (using promotion as explained in Section 2.1) and is intended to be a complete specification of *an* editor with each one being successively closer to UQ2.

The modularity project has focussed on the fourth stage since it incorporates the core editor functionality but omits the file and window handling. To give some idea of the size of the specification, the first four stages have 271 schema definitions, of which only 32 are top-level operations for the fourth level. The state contains 54 state variables, a number that increases to 74 after expansion of schema-valued variables, including promotions. Schema expansion is performed, taking into account the Z conventions for 'Δ' and 'Ξ' schemas. Expressions are expanded using the semantics of the schema operators. As a result of expansion, operation schemas have large signatures: 50 variables is typical in the case study.

The Z definitions are analysed and annotated with attributes:

- Each state schema has an *invariant* attribute that lists all variables linked by a state invariant. There are 165 invariant attributes in the case study (some of these are generated by the expansion of schema-valued variables). A functional dependency attribute records where one set of variables completely determine the value of some other variable. These attributes identify derived variables and elements of the state that may be modified implicitly by operations.

- Each operation schema has three attributes that contain a list of variables that are read, written, or are explicitly unchanged by the operation. These lists are disjoint. The case study has 69 read, 1281 write and 1012 equate dependencies.

The analysis generates relations between elements of the specification state and the system operations. One compact display for this information is a table, with each row representing an operation and each column representing a state variable. The entries in the table encode the type of relationship that holds between the operation and the state variable. Figure 2 is an example representing part of the results for the case study after applying some clustering techniques. The figure gives an indication of the results obtained without details from the editor case study of the particular state variables. Only the *read* and *written* attributes are

Operations		State Variables																																			
											1	1	1	1	1	1	1	1	1	1	2	2	2	2	2	2	2	2	2	2	3	3	3	3	3	3	3	3	3
		1	2	3	4	5	6	7	8	9	0	1	2	3	4	5	6	7	8	9	0	1	2	3	4	5	6	7	8	9	0	1	2	3	4	5	6	7	8
highlight	1	r	r	r	r					r	r			w	w	w	w	w	w	w	w	w	w	w	w														w
operations	2	r	r	r	r					r	r			w	w	w	w	w	w	w	w	w	w	w	w														w
	3				r					r	r			w	w	w	w	w	w	w	w	w	w	w	w														
	4				r					r	r			w	w	w	w	w	w	w	w	w	w	w	w														
	5								r	r	r			w	w	w	w	w	w	w	w	w	w	w	w														
	6								r	r	r			w	w	w	w	w	w	w	w	w	w	w	w														
	7								r	r	r			w	w	w	w	w	w	w	w	w	w	w	w														
	8								r	r	r			w	w	w	w	w	w	w	w	w	w	w	w														
	9								r	r	r			w	w	w	w	w	w	w	w	w	w	w	w														
	10							r	r	r	r			w	w	w	w	w	w	w	w	w	w	w	w														
pan forward	11									r	w	w	w	w	w	w	w	w	w	w	w	w	w	w	w													w	
pan backward	12									r	w	w	w	w	w	w	w	w	w	w	w	w	w	w	w													w	
zoom-in	13							r		r	w	w	w	w	w	w	w	w	w	w	w	w	w	w	w														
zoom-out	14	r	r	r	r	r	r	r	r	r	w	w	w	w	w	w	w	w	w	w	w	w	w	w	w														
fold	15	w	w	w	w	w	w	w	w	w	r		w	w	w	w	w	w	w	w	w	w	w	w	w			w											
unfold	16	w	w	w	w	w	w	w	w	w	r		w	w	w	w	w	w	w	w	w	w	w	w	w				w										
search	17	w	w	w	w	w	w	w	w	w	w	w	w	w	w	w	w	w	w	w	w	w	w	w	w									r	w	w	w		
operations	18	w	w	w	w	w	w	w	w	w	w	w	w	w	w	w	w	w	w	w	w	w	w	w	w									r	w	w	w		
	19	w	w	w	w	w	w	w	w	w	w	w	w	w	w	w	w	w	w	w	w	w	w	w	w								r	r	w	w	w		
	20	w	w	w	w	w	w	w	w	w	w	w	w	w	w	w	w	w	w	w	w	w	w	w	w							w	r	w	w	w	w		
	21									r			w	w	w	w	w	w	w	w	w	w	w	w						w	w	w	w	w					
paste	22	w	w	w	w	w	w	w	w	w	w	w	w	w	w	w	w	w	w	w	w	w	w	w	w	w	w												
operations	23	w	w	w	w	w	w	w	w	w	w	w	w	w	w	w	w	w	w	w	w	w	w	w	w	w	w												
	24	w	w	w	w	w	w	w	w	w	w	w	w	w	w	w	w	w	w	w	w	w	w	w	w	w	w												
	25	w	w	w	w	w	w	w	w	w	w	w	w	w	w	w	w	w	w	w	w	w	w	w	w	w	w												
	26	w	w	w	w	w	w	w	w	w	w	w	w	w	w	w	w	w	w	w	w	w	w	w	w	w	w												
	27	w	w	w	w	w	w	w	w	w	w	w	w	w	w	w	w	w	w	w	w	w	w	w	w	w	w												
insert	28	w	w	w	w	w	w	w	w	w			w	w	w	w	w	w	w	w	w	w	w	w	w	w													
backspace	29	w	w	w	w	w	w	w	w	w			w	w	w	w	w	w	w	w	w	w	w	w	w	w													
zoom-out+insert	30	r	r	r	r	r	r	r	r	r	w	w	w	w	w	w	w	w	w	w	w	w	w	w	w	w	w												
save	31	w	w	w	w	w	w	w	w	w	w		w	w	w	w					r	w	w	w	w	w	w												
set language	32						r			w												w	w	w			w												

Figure 2: Example relationship table between state variables and operations for UQ2.

shown to highlight the structure of references. This table also omits dependency and invariant relationships as these seem to be of secondary importance in finding module structure.

To illustrate how the table may be interpreted, the first 23 variables represent various aspects of the editor contents (e.g. 5 is the parse tree, 6 is the string of characters, and 24 is the currently highlighted string); 26 is the current input language; variables 29 to 36 are search parameters. The first ten operations involve moving the highlight. Some components of editor contents are potentially changed by these operations since highlight moving operations can change the current context which is part of the editor contents. Operations 17 to 21 are search operations while the next nine operations perform insertions. The cross-reference table generated by the analysis phase has been useful for checking the formal specification. Several errors and omissions have been identified by tracing relationships identified by the analysis phase.

The evaluation of the various synthesis algorithms is at an early stage. Our preliminary results are encouraging based on informal evaluations using knowledge of the specification and the current implementation. We need to establish methods for assessing the quality of module designs.

For the case study, the split algorithm is more appropriate than the join algorithm as there are a lot of highly dependent variables and because we expect to get only a few modules at the top level. Many iterations of the join algorithm are required to achieve a design with a satisfactory level of coupling.

We are interested in comparing the structure of modules arising from analysing the formal specification of the UQ2 editor with those developed by conventional software development methods. We expect that such a comparison will identify additional strategies employed by human designers, but may reveal opportunities for modularisation that were not obvious.

4 Conclusions and future work

We have presented techniques for analysing large specifications with a view to devising a modular structure for an implementation, and discussed the application of these techniques to a substantial case study. The results are encouraging.

The techniques discussed in this paper are seen as aids to the designer, rather than a prescription of a mechanical process for modularisation. Ultimately the designer must judge the best approach to system design, aided by the information provided by the techniques.

A partial implementation of the analysis process as described in Section 2.1 has been undertaken using Nu-Prolog [17]. The prototype handles analysis from the level of specifications down to the level of schemas, but beyond that relies on data obtained from the specification by hand. Instead of operating on the abstract syntax tree of a specification, the prototype works on a collection of definitions derived manually from a given specification. Various prototypes for the synthesis phase have been developed which use data generated from the analysis phase.

Our goals for future work include

1. completing the tool set for the analysis and synthesis phases. For the analysis phase, direct extraction of the information from a Z specification document is required. Developing tools that allow some interaction in the synthesis process would be desirable so

the designer can influence the modular decomposition. This capability would allow a designer to experiment with alternative decompositions and observe the consequences.

2. evaluating more fully the results obtained with automatic analysis of Z specifications and comparing them with traditional approaches to modular decomposition. Additional case studies will be helpful to demonstrate the applicability of our techniques.

3. integrating the tools for module design into a user-oriented environment. Our favoured approach is to use a language-based editor customised for Z documents (such as [7]) and couple the modularity tools as 'back-end' analysers [19, 20].

4. investigating the projection of operations onto modules to generate possible signatures and pre- and post-conditions to describe required behaviour.

For application in a real software development environment, the development of tools to assist in the application of these techniques is essential.

5 Acknowledgements

We gratefully acknowledge feedback on the above work from Peter Bancroft, Roger Duke, Cliff Jones, Brendan Mahony, Nigel Ward and Luke Wildman. The work reported in this paper has been supported by Australian Research Council grant number A48931426: *Modularity in the derivation of verified software.*

References

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

[2] D. Carrington, D. Duke, R. Duke, P. King, G. Rose, and G. Smith. Object-Z: An object-oriented extension to Z. In S. Vuong, editor, *Formal Description Techniques (FORTE'89).* North Holland, 1990.

[3] S.C. Choi and W. Scacchi. Extracting and restructuring the design of large systems. *IEEE Software*, 7(1):66–71, January 1990.

[4] R. Duke, P. King, G. Rose, and G. Smith. The Object-Z specification language: Version 1. Technical Report 91–1, Software Verification Research Centre, Department of Computer Science, University of Queensland, Australia, 1991.

[5] D.W. Embley and S.N. Woodfield. Assessing the quality of abstract data types written in ADA. In *Proceedings of the 10th International Conference on Software Engineering*, pages 144–153. IEEE, 1988.

[6] A. Hall. Seven myths of formal methods. *IEEE Software*, 7(5):11–19, 1990.

[7] I. Hayes, R. Neucom, and J. Welsh. An editor for Z specifications. In *Advance Papers for CASE '89 Workshop, London*, July 1989.

[8] Ian Hayes, editor. *Specification Case Studies.* Prentice Hall International, second edition, 1993.

[9] C. B. Jones. *Systematic Software Development Using VDM.* Prentice Hall, second edition, 1990.

[10] B. Meyer. *Object-Oriented Software Construction.* Prentice Hall, 1988.

[11] C. Morgan. *Programming from Specifications.* Prentice Hall, 1990.

[12] C. Morgan and B. Sufrin. Specification of the UNIX filing system. *IEEE Transactions on Software Engineering*, 10(2):128–142, 1984.

[13] D.L. Parnas. On the criteria to be used in decomposing systems into modules. *Communications of the ACM*, 15(12):1053–1058, December 1972.

[14] S. Patel, W. Chu, and R. Baxter. A measure for composite module cohesion. In *Proceeding of the 14th International Conference on Software Engineering*, pages 38–48, 1992.

[15] I. Sommerville. *Software Engineering.* Addison-Wesley, second edition, 1985.

[16] J. M. Spivey. *The Z Notation: A Reference Manual.* Prentice Hall, second edition, 1992.

[17] J. Thom and J. Zobel. *NU-Prolog reference manual (version 1.3).* Department of Computer Science, University of Melbourne, 1986.

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

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

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

[21] E. Yourdon and L.L. Constantine. *Structured Design: fundamentals of a discipline of computer program and systems design.* Yourdon Press, second edition, 1978.