An Editor for Refinement Structure Diagrams

Lindsay Groves and Ivan Tuckwell

School of Mathematical and Computing Sciences
Victoria University of Wellington
Wellington, New Zealand

Abstract. In this paper, we discuss the design of a refinement tool, based on Refinement Structure Diagrams, intended to support a mixture of formal and informal program development. Refinement Structure Diagrams are much like traditional structure diagrams with formal specifications attached to the nodes. Subcomponents of a Refinement Structure Diagram can then be seen as representing the application of refinement rules in a formal development. The tool allows structure diagrams to be edited in a fairly conventional way, for example, adding statements to a sequential decomposition or adding a branch to an **if** statement. Formal specifications can be provided when constructs are created, leading to a strictly formal top-down development; or they can be omitted and added later as required. Thus, we can formally verify as much or as little of the program development as we choose. The key to providing this flexibility is in the use of metavariables, which allow us to defer the binding of parameters in refinement rules. We discuss the way in which metavariables are handled in the tool, and how the style of operation supported by the tool affects the forms of refinement rules used in the tool.

1 Introduction

The refinement calculus ([Back, 1988], [Morgan, 1994], [Morgan and Vickers, 1994]) is usually presented as a formalisation of the stepwise refinement approach to program design [Wirth, 1971]. Most discussions of program refinement, and most existing refinement tools, consequently assume (often implicitly) a methodology whereby the programmer turns a formal specification into an executable program via a sequence of transformations, discharging any proof obligations required to ensure that correctness is preserved.

While top-down design may be a good ideal to strive for, it does not reflect the way in which programmers actually construct programs [Worden, 1991]. In practice, programmers explore many alternative design options and switch between high-level and low-level concerns, often looking a long way ahead to anticipate details of data structure implementations or to predict the impact of possible design decisions. When these predictions are imperfect, the programmer will often go back and change earlier decisions, and adjust other parts of the program accordingly. A top-down design should thus be seen as a good way of recording and explaining a program development [Parnas and P. C. Clements, 1986], describing the way in which the programmer would like to have constructed the program (after eliminating the mistakes and dead-ends), rather than recording what actually happened during the design process.

This is highlighted further when we consider formal software development. It has been argued strongly (e.g. [Dijkstra, 1976]) that design and proof should go hand-in-hand, so that design steps are verified as decisions are taken and verification concerns can guide the design process. The use of formal development methods, such as the refinement calculus, should thus help to make the design process more systematic, for example by eliminating dead-ends that lead to incorrect programs. We should, however, still expect programmers to look ahead to consider ease, efficiency and even feasibility of later implementation, and to change their minds and try different approaches when things don't work out as expected. It is also unreasonable to expect the various alternatives considered to be worked out completely formally.

This issue becomes more important when we consider the design of a refinement tool. If the tool is too rigid in the methodology it supports, the programmer will be forced to work out most of the details of a derivation with pencil and paper *before* beginning to use the tool. If the tool is going to *help* the programmer to design the program, rather than just checking the design, the programmer must be able to try out ideas in an informal or semi-formal way before undertaking a full formal development, and must be able to change earlier decisions without having to laboriously undo and redo steps that are not affected by the change.

These considerations suggest that we should look for ways of making a formal development methodology less rigid, and attempt to design tools to support this more flexible methodology. We would like to be able to support these kinds of development patterns within the same framework as the idealised top-down development, so that the advantages of a top-down development are obtained where appropriate.

Taking a broader view of the role of formal methods in software development, we would like to support a more flexible approach still. The use of formal methods tends to be seen as an all or nothing choice, which creates huge barriers to the adoption of formal methods. There is growing evidence that use of formal methods in selected parts of an application can have significant benefits at reasonable cost [Larsen, Fitzgerald and Brookes, 1996]. By extending the flexible approach outlined above, we believe we can also accommodate a development process in which a program is written completely informally and a complete verification done subsequently, or in which only selected parts of the program are formally specified and (perhaps) verified.

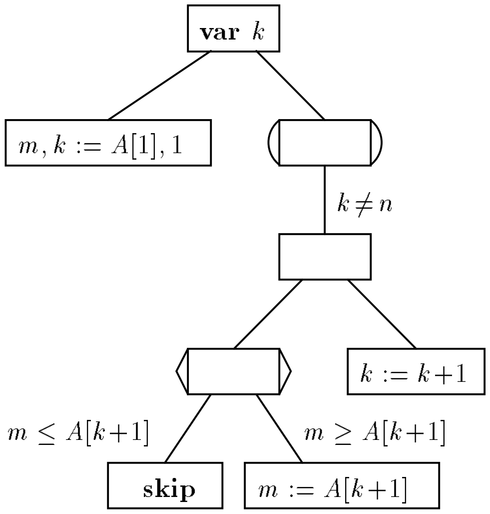
In this paper, we describe a refinement tool in which we attempt to support a mixture of formal and informal development in a flexible manner. The tool is based on Refinement Structure Diagrams [Groves, 1996a], which are much like conventional structure diagrams (e.g. [Jackson, 1975], [Doran and Tate, 1972]), with formal specifications attached to the nodes. Subcomponents of a Refinement Structure Diagram can then be seen as representing the application of refinement rules in a formal development.

Section 2 describes the structure diagrams we use, and explains how they can be used as the basis for the flexible methodology outlined above. Section 3 then describes the editing operations provided in our prototype editor, which allows it to support this methodology, and Section 4 discusses some additional aspects of the editor's design. Finally, Section 5 discusses some of the strengths and weaknesses of the current prototype and outlines some planned extensions.

2 Refinement Structure Diagrams

As discussed above, we wish to be able to mix formal and informal program development, and to allow the formal parts of a program development to be constructed in a flexible manner. To do this, it must thus be possible to separate decisions about program structure (introducing a sequence, **if**, **do**, etc.) from decisions about formal specifications and loop invariants, and proof of verification conditions. We achieve this by using a form of structure diagram, in which formal specifications (and other formal annotations) may be attached to nodes [Groves, 1996a], and the steps taken in constructing a diagram can be understood as corresponding to refinement steps in a formal derivation.

Our structure diagrams are similar to the traditional ones of Jackson, etc. In a complete structure diagram, the leaf nodes represent primitive statements and the non-leaf nodes represent control structures. We use rectangular boxes for sequential composition and primitive statements, angled boxes for choice, and rounded boxes for loops. Our program design language is based on Dijkstra's guarded command language [Dijkstra, 1976], so choice and loop nodes may have an arbitrary number of children, with guards attached to the edges. For example, a structure diagram depicting the design of a program to find the minimum of a non-empty array segment $A[1..n]$ is shown on the left of Figure 1; the corresponding program in Dijkstra's guarded command language is shown on the right.



$$
\begin{array}{l}
[\![\ \mathbf{var}\ k \bullet \\
\quad m, k := A[1], 1; \\
\quad \mathbf{do}\ k \neq n \rightarrow \\
\qquad \mathbf{if}\quad m \leq A[k+1] \rightarrow \mathbf{skip} \\
\qquad [\,]\quad m \geq A[k+1] \rightarrow m := A[k+1] \\
\qquad \mathbf{fi}; \\
\qquad k := k+1 \\
\quad \mathbf{od} \\
]\!]
\end{array}
$$

Fig. 1. Structure Diagram and Program for Array Minimum

A non-leaf node (which may be a leaf in an incomplete structure diagram) can also be seen as representing a procedural abstraction — a description of an action to be performed, which (when the diagram is complete) is implemented by the descendants of that node. These abstractions are usually described in English (or some other natural language), or omitted when deemed obvious.[1]

Because these procedural abstractions are not formally specified in a normal structure diagram, it is not possible to formally verify design steps until the whole program has been constructed. To allow a structure diagram to record a formal derivation, and allow design steps to be verified in a top-down fashion, we extend our structure diagrams, to allow formal specifications (written as specification statements) to describe these procedural abstractions. We call the resulting diagrams Refinement Structure Diagrams, though when the intention is clear we talk about structure diagrams or just diagrams instead.

We adopt a few conventions to allow Refinement Structure Diagrams to be displayed compactly. For example, we elide a primitive statement and the abstraction it implements into a single node; we also elide a declaration or syntactic definition with its parent. Adding these annotations to the structure diagram in Figure 1, we obtain the Refinement Structure Diagram shown in Figure 2.

While this diagram looks quite cluttered on first glance, it in fact embodies the same information as a formal derivation in a much more compact way. We can understand a (partial) structure diagram as representing a (partial) program derivation. Each one-level subtree corresponds to the application of a refinement rule, so we can judge whether a structure diagram represents a valid derivation in the same way we do for a normal derivation in the refinement calculus (for details see [Groves, 1996a]). For example, Figure 3 shows, in a more conventional way, the steps corresponding to the top four nodes in this diagram.

Having introduced both (plain) structure diagrams, where procedural abstractions are specified informally, and refinement structure diagrams, where they are specified formally, we can now mix the two, and allow diagrams where only some of the abstractions are specified formally. This range of formality then allows us to accommodate a wide range of development patterns based on the same representation. For example, we might:

[1] We omit these descriptions in the above example, since we are primarily concerned with illustrating the structure of the diagrams used.

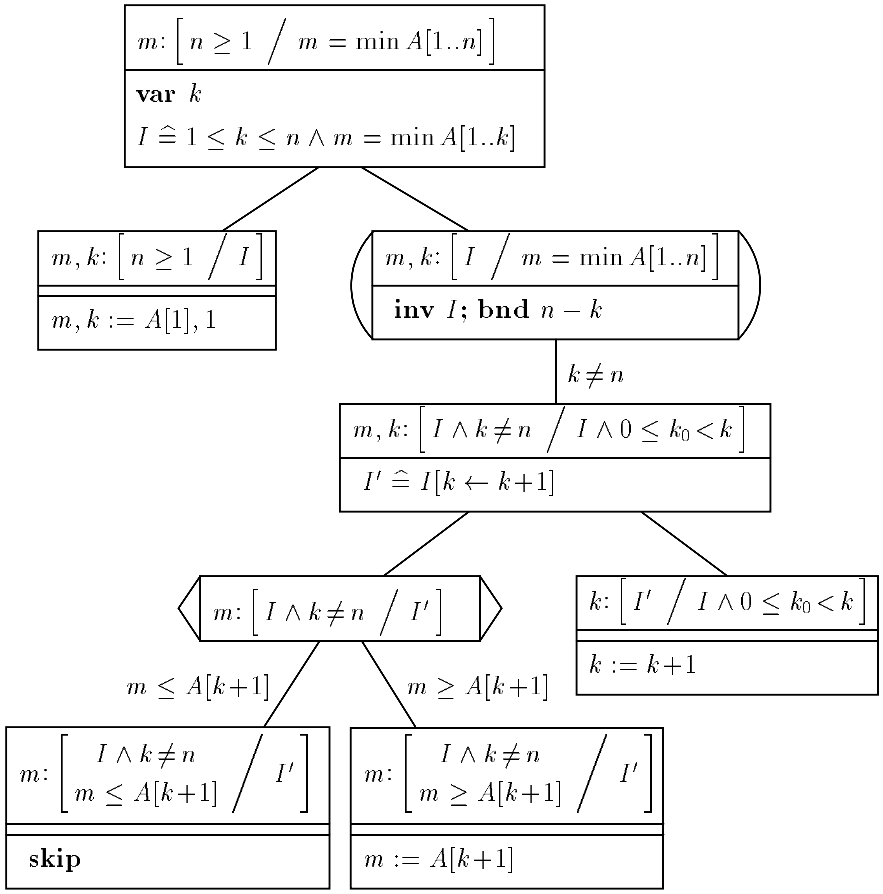


Fig. 2. Refinement Structure Diagram

- Construct a conventional structure diagram, omitting formal specifications completely, as in Figure 1.
- Construct a completely formal refinement structure diagram, providing formal specifications and discharging proof obligations for all nodes as they are created. This might result in a diagram as shown in Figure 2.
- Construct a completely formal diagram, but do so in an order where we sometimes build several steps informally and then fill in the formal details. This might also result in a diagram as shown in Figure 2, but via a different sequence of steps.
- As an extreme version of the above, we might construct the whole diagram informally, and the fill in the formal details later. This would make sense if we already had the program and wanted to construct a formal derivation as a way of verifying it (cf. [Groves, 1996c]).
- Provide formal specifications (and prove the necessary conditions) for selected parts of the diagram. For example, in the development presented in Figure 2, we might choose to only prove that the loop body preserves the invariant, or that the loop terminates.

In order to accommodate some of these variations, we need to be able to partially specify pre- and postconditions, loop invariants etc. We do this by using *metavariables* as “placeholders” for formulas, expressions, etc. that have not been chosen yet, and allowing values to be inserted later.

$$m\colon \left[\, n \geq 1 \;\middle/\; m = \min A[1 \mathinner{..} n] \,\right] \qquad (i)$$

$$\begin{array}{rcll}
(i) & \sqsubseteq & \textit{(Introduce Local Variable)} & \\
 & & \mathbf{var}\ k \bullet & \\
 & & m, k\colon \left[\, n \geq 1 \;\middle/\; m = \min A[1 \mathinner{..} n] \,\right] & (ii) \\
(ii) & \sqsubseteq & \textit{(Split Specification)} & \\
 & & m, k\colon \left[\, n \geq 1 \;\middle/\; 1 \leq k \leq n \wedge m = \min A[1 \mathinner{..} k] \,\right]; & (iii) \\
 & & m, k\colon \left[\, 1 \leq k \leq n \wedge m = \min A[1 \mathinner{..} k] \;\middle/\; m = \min A[1 \mathinner{..} n] \,\right] & (iv) \\
(iii) & \sqsubseteq & \textit{(Introduce Assignment)} & \\
 & & m, k := A[1], 1 & \\
(iv) & \sqsubseteq & \textit{(Introduce DO)} & \\
 & & \mathbf{do}\ k \neq n \rightarrow & \\
 & & \qquad m, k\colon \left[\begin{array}{c} 1 \leq k < n \\ m = \min A[1 \mathinner{..} k] \end{array} \middle/ \begin{array}{c} 1 \leq k \leq n \\ m = \min A[1 \mathinner{..} k] \\ k_0 < k \end{array} \right] & \\
 & & \mathbf{od} &
\end{array}$$

Fig. 3. Beginning of Conventional Refinement

This use of metavariables is central to the design of our editor, and is discussed in more detail in Section 4.

3 Editing Refinement Structure Diagrams

We now consider the kinds of actions required in an editor for Refinement Structure Diagrams that will support the kinds of development styles outlined above, and outline how these facilities are provided in our prototype editor.

We want to be able to support both formal and informal development, and be able to do the formal part top-down, bottom-up, or in any other order that makes sense to the programmer. We need operations to add nodes to the current diagram corresponding to procedural abstractions, program constructors or primitive statements. We must be able to add formal and/or informal specifications to procedural abstractions, edit the contents of nodes, and delete nodes. We also need to be able to generate proof obligations so that the correctness of the design may be verified.

We also want the user interface to be simple enough that the range of development styles outlined all appear fairly "natural", so the user will feel comfortable adopting whatever style suits their preference and requirements. To achieve all this with a fairly simple interface, we provide three sets of editing operations.

The first set are simple structure-oriented editing operations, allowing the user to add a parent, child, or left or right sibling to a node, or delete a node (and its children). The nodes added are shown as generic, empty, rectangular boxes — other operations are used to add text to them or indicate that they represent particular statement types (other than sequence).

The second set of operations allow the user to edit various annotations associated with a node. In particular, the user can:

- Add informal, descriptive text to a node, or edit existing text.

- Add a formal specification (specification statement) to a node, or edit an existing specification. This may bind metavariables, whose values are then propagated automatically to other occurrences.
- Edit a guard associated with a child of an **if** or **do** node.
- Add a syntactic definition to a node, defining a symbol that can then be used later as an abbreviation, or edit an existing definition.

The third set of operations are used to perform "refinements", and affect the statement type associated with a node. In particular, the user can:

- Add an assignment or **skip** statement to a node.
- Add variable or logical constant declarations to a node.
- Expand a node as a sequence, adding a specified number of children. The children are given specifications $w\colon[\,\Phi\ /\ \Omega_1\,]$, $w\colon[\,\Omega_1\ /\ \Omega_2\,], \cdots, w\colon[\,\Omega_{n-1}\ /\ \Psi\,]$, where w, Φ and Ψ are the frame, precondition and postcondition of the parent's specification, and $\Omega_1, \cdots, \Omega_{n-1}$ are metavariables standing for intermediate assertions, yet to be provided.
- Mark a node as being an **if** or **do**. This changes the shape of the node, to angled for **if** and rounded for **do**. In the case of a **do**, entries are added for the loop invariant, bound function and the logical constant used to refer to the initial value of the bound function. No children are added by this operation,[2] but any existing children will have guards attached, as will any children added subsequently. The guards are represented by metavariables, and can be inserted at any stage.
- Undo the refinement applied to this node, in which case it reverts to a plain rectangular box.

Many of these operations can be applied to a node anywhere in the diagram. Thus, for example, it is easy to add a variable declaration to a node near the top of the diagram, or modify an intermediate assertion or loop invariant at some earlier point. The selection of options made available to the user is sensitive to the current node type; for example, an assignment of **skip** can only be added to a leaf node, and an **if** or **do** can't be turned into a sequence.

There are also buttons controlling global operations to begin a new diagram, load/save a diagram to/from a file, display the code corresponding to the current diagram (English annotations are included as comments), display the proof obligations associated with the current diagram, or exit from the editor. Both code and proof obligations can be saved to a file.

4 Editor Design

We will now discuss briefly two additional aspects of the internal design of the editor.

4.1 Metavariables

A key aspect of the editor's design is the use of metavariables. Every node in a structure diagram has a specification associated with it, and any component that has not been specified is represented by an unbound metavariable. Thus, not having a specification in fact corresponds to having an "unspecified" specification. Our use of metavariables was motivated by Ray Nickson's work ([Nickson, 1994], [Nickson and Groves, 1994]), but takes the idea further.

When a refinement operation is performed (e.g. introducing a sequential composition, or adding a child to an **if** or **do** node), any parameters in the refinement rule (i.e. pattern variables whose

[2] It would be more consistent to give the user the option of specifying the number of branches at this point, as is done for a sequence.

values are not determined by matching against the specification being refined, which includes intermediate assertions, loop invariants and guards) are left as metavariables.

Subsequent editing may provide values for metavariables, which are then propagated through all places where the metavariable occurs. A metavariable can only be bound by editing its initial occurrence.

Leaving all of the metavariables (except guards) unbound corresponds to constructing a plain structure diagram with no specifications. Providing values for metavariables as soon as they are introduced corresponds to a strict top-down formal development. Adding values for metavariables at other times corresponds to the more flexible development patterns discussed above.

In order to maintain the illusion that a node with an unspecified specification has no specification, the editor currently uses a rough and ready heuristic to determine when to display the specification: If none of the metavariables in the specification are bound, the specification is not displayed, otherwise it is displayed. This is not ideal, as it leads to some odd behaviour, for example, displaying a specification such as:

```
:[<Guard2> and <Pre1>, <Post1>]
```

because the precondition is "bound" to a conjunction of two unbound metavariables.

4.2 Generalised rules and Defaults

The usual rules used in the refinement calculus (e.g. [Morgan, 1994]) are carefully tuned for top-down development. For example, the usual rule to introduce an **if** looks something like:

$$\frac{P \Rrightarrow \bigvee_i B_i}{\tilde{w}\colon [\; P \;/\; Q \;] \;\sqsubseteq\; \textbf{if} \;\; \|_i B_i \rightarrow \tilde{w}\colon [\; P \wedge B_i \;/\; Q \;] \;\; \textbf{fi}}$$

This rule relies on the fact that in a top-down development, the specifications for the branches of an **if** can be constructed from the specification being refined and the guards. This reduces the number of proof obligations required.

If we are going to allow specifications to be added to nodes in a structure diagram in a more flexible way (e.g. we might add specifications to the branches of an **if** before giving a specification to its parent), we cannot rely on this relationship. In this case, we need to use a more general form of rule:

$$\frac{\bigwedge_i \tilde{w}_i \subseteq \tilde{w}, \;\; \bigwedge_i P \wedge B_i \Rrightarrow P_i, \;\; \bigwedge_i Q_i \Rrightarrow Q, \;\; P \Rrightarrow \bigvee_i B_i}{\tilde{w}\colon [\; P \;/\; Q \;] \;\sqsubseteq\; \textbf{if} \;\; \|_i B_i \rightarrow \tilde{w}_i\colon [\; P_i \;/\; Q_i \;] \;\; \textbf{fi}}$$

While the generalised rule is more flexible, it is also more troublesome to apply, since it requires separate pre- and postconditions to be specified for each branch. To avoid having to do this, when children are added to an **if** or **do** node, they are automatically given the pre- and postconditions that would be given by the "top-down" version of the rule. Thus, in the above rule, $P \wedge B_i$ would be used as the default binding for P_i and Q as the default for Q_i.

Also, if a node already has children when it is marked as an **if**, and those children have no specifications (their specifications are unbound), an extra menu option *Use Default Spec* becomes available, allowing the default specification to be inserted.

This use of defaults provides an agreeable compromise between the flexibility of the general rule and the ease of use of the top-down rule.

5 Conclusions and Future Work

Our current prototype was developed as a basis for trying out and further developing the ideas outlined in [Groves, 1996a] and [Groves, 1996c]. As a proof-of-concept has been quite successful, though there are clearly many improvements and extensions that could be made.

The most glaring shortcomings are the lack of formatting for specifications, and some awkwardness in the use of pull-own menus activated using the mouse button.[3] The editor also only produces proof obligations for the entire diagram, rather than for selected nodes or subtrees, and provides no support for doing proofs.

There is also clearly a need for some way to break a structure diagram up into several smaller diagrams, and "call" them as one would call procedures or macros (see [Groves, 1996a] and [Groves, 1996b]). It should also be possible to maintain a library of structure diagrams that can be used as components, either by calling them in this way or inserting them directly into the current diagram.

The editor currently only supports refinement rules that turn specifications into primitive statements (assignment or **skip**), control structures (sequence, **if** or **do**), or other specifications (we have a single rule that combines the effects of the usual *Weaken precondition*, *Strengthen postcondition* and *Contract frame* rules) or introduce declarations (for local variables and logical constants). It does not support any of the "structure modifying" rules found in [Morgan, 1994], such as *Skip composition*, *Collapse identical branches*, or *Left distribute composition over alternation*. To support such rules we would need to find a way of displaying structure diagrams so as to show that a whole subtree had been replaced; some ideas for this are presented in [Naftalin, 1994].

An interesting question raised by this work is just what need there is for such rules. Our editor presents the view that what a programmer is doing in formal software development is editing a formal derivation, not simply constructing it. Thus, we can freely change earlier steps in a derivation, to obtain the desired final outcome, not simply add steps to a sequential design history. If we end up with a **skip** that can be eliminated, identical branches in an **if**, or common tails in the branches of an **if**, this might be taken as indicating that an earlier decision was wrong. It is easy to delete a node that has been refined to a **skip**, and we probably just end up with a slightly different proof obligation. The effect of *Collapse identical branches* or *Left distribute composition over alternation* (and other similar rules) can also be obtained by editing earlier steps, though some adjustment will be required to earlier specifications in order to make the proofs work. There are some situations where we would not want to revise the derivation in this way, for example, where we want to simplify the program after a data refinement [Groves, 1994] (which is something else we have not yet considered), but these appear to be fairly rare. Ultimately, the user should have the choice to record the derivation in whatever way they prefer.

References

[Back, 1988] R. J. R. Back. "A calculus of refinements for program derivations". *Acta Informatica* **25** (1988), pp593–624.

[Dijkstra, 1968] E. W. Dijkstra. "A constructive approach to the problem of program correctness". *BIT* **8** (1968), pp174–186.

[Dijkstra, 1975] E. W. Dijkstra, "Guarded commands, nondeterminacy and formal derivation of programs", *Communications of the ACM*, **18** (August 1975), pp453–457.

[Dijkstra, 1976] E. W. Dijkstra. *A Discipline of Programming*. Academic Press, 1976.

[Doran and Tate, 1972] Bob Doran and Graham Tate. *An Approach to Structured Programming, Parts I and II*. Massey University Computer Unit, Publications No. 6 and 9, June 1972.

[3] The editor is implemented in SICSTUS Prolog, using tck/tk for the graphics; this was our first adventure in using the Prolog-tck/tk combination to implement an interactive application.

[Groves, 1994] Lindsay Groves. "Deriving language recognition algorithms: A case study in combining program specialisation and data refinement". *Proc. 6th Refinement Workshop*, British Computer Society, London, January 1994.

[Groves, 1996a] Lindsay Groves. "Program Refinement with Structure Diagrams". Fifth Australasian Refinement Workshop, University of Queensland, April 1996. (Victoria University of Wellington, Technical Report CS-TR-96/5.)

[Groves, 1996b] Lindsay Groves. "Procedures in the Refinement Calculus: A New Approach?". *Proc. 7th Refinement Workshop*, Springer-Verlag, 1996.

[Groves, 1996c] Lindsay Groves. "Program Comprehension with the Refinement Calculus". *Proc. 2nd Program Comprehension Workshop*, Durham, July 1996. (Victoria University of Wellington, Technical Report CS-TR-96/21.)

[Jackson, 1975] M. A. Jackson. *Priciples of Program Design*. Academic Press, 1975.

[Larsen, Fitzgerald and Brookes, 1996] P. G. Larsen, J. S. Fitzgerald and T.M. Brookes. "Applying Formal Specification in Industry". *IEEE Software*, **13**, 3 (May 1996), pp 48-56

[Morgan, 1988a] Carroll Morgan. "The specification statement". *TOPLAS* **10**, 3 (July 1988). Also in [Morgan and Vickers, 1994], pp7–30.

[Morgan, 1994] Carroll Morgan. *Programming from Specifications*. Prentice-Hall, Second Edition, 1994.

[Morgan and Vickers, 1994] Carroll Morgan and Trevor Vickers (Eds). *On the Refinement Calculus*. Springer-Verlag, 1994.

[Naftalin, 1994] M. Naftalin. "Informal Strategies in Design by Refinement". *Proc. Formal Methods Europe 1994*, pp401–420.

[Nickson, 1994] R. G. Nickson. *Tool Support for the Refinement Calculus*. PhD thesis, Victoria University of Wellington, 1994.

[Nickson and Groves, 1994] Raymond G. Nickson and Lindsay J. Groves. "Metavariables and Conditional Refinements in the Refinement Calculus". *Proc. 6th Refinement Workshop*, Springer-Verlag, 1994, pp167–187.

[Parnas and P. C. Clements, 1986] D. L. Parnas and P. C. Clements. "A Rational Design Process: How and Why to Fake It". *IEEE Transactions on Software Engineering*, **SE-12**, 2 (February 1986), pp. 251-257. (Also in *Great Papers in Computer Science*, edited by Phillip Laplante, West Publishing Co, 1996, pp. 442-451.)

[Wirth, 1971] N. Wirth. "Program development by stepwise refinement". *CACM* **14** (April 1971), pp 221–227.

[Worden, 1991] Robert Worden. "The Process of Refinement". *Proc. 4th Refinement Workshop*, J. M. Morris and R.C. Shaw (Eds.), Springer Verlag, 1991, pp 1–5.