VICTORIA UNIVERSITY OF WELLINGTON
Te Whare Wananga o te Upoko o te Ika a Maui

Department of Computer Science

PO Box 600
Wellington
New Zealand

Tel: +64 4 471 5328
Fax: +64 4 495 5232
Internet: Tech.Reports@comp.vuw.ac.nz

Program Comprehension with the Refinement Calculus

Lindsay Groves

Technical Report CS-TR-96/21
October 1996

Abstract

Formal program derivations are difficult to understand because of the amount of detail that needs to be presented. We discuss a way of presenting program derivations in the refinement calculus, based on a form of structure diagram, annotated with formal specifications and details of refinement steps. We also show that these diagrams can be used as a basis for constructing formal derivations for existing programs.

Publishing Information

Second UK Workshop on Program Comprehension, Durham, July 1996.

Program Comprehension with the Refinement Calculus

Lindsay Groves

Department of Computer Science
Victoria University of Wellington
Wellington, New Zealand
Email: `lindsay@comp.vuw.ac.nz`

Abstract

Formal program derivations are difficult to understand because of the amount of detail that needs to be presented. We discuss a way of presenting program derivations in the refinement calculus, based on a form of structure diagram, annotated with formal specifications and details of refinement steps. We also show that these diagrams can be used as a basis for constructing formal derivations for existing programs.

1 Introduction

Program comprehension is an essential component of many software development activities, including teaching, debugging, maintenance, reuse and re-engineering. This has lead to a number of approaches to developing software tools and techniques to aid in program comprehension, addressing comprehension of different aspects of programs and seeking to aid comprehension in different ways. Some approaches aim to help a user to understand the structure and/or behaviour of a program, usually by presenting the program and/or its behaviour in a suitable form and providing facilities for inspection and navigation. Other approaches aim to construct a description of the program in a more convenient or more intelligible form. This may be a better structured or more abstract version of the program, a natural language description or a formal specification, and may be quite different from the original program. A tool might attempt to construct this description automatically, using heuristic techniques, or might assist a user in constructing this description interactively.

This paper addresses some issues of program comprehension that arise in the context of formal software development. In this setting, we are interested in understanding what a program does, which is expressed in terms of a formal specification, and how it does it, which is expressed in terms of a formal derivation of the program from this specification. Formal derivations are necessarily quite complex, and need to convey a large amount of detailed information, so we wish to be able to present derivations in a way that makes them easy for the programmer to understand and modify. Since we wish to be able to use formal derivation techniques with existing programs, we also wish to be able to construct a derivation, and perhaps also a specification, for a program that was not developed using formal techniques.

We are working in the context of the refinement calculus, so a specification is a program in a wide-spectrum language, which need not be executable, and a derivation is a sequence of refinement rule applications which transforms the specification into the resulting executable program. The key to our approach is to present a derivation as a form of structure diagram, annotated with formal specifications and details of refinement steps. We claim that presenting derivations in this way makes them easier to understand than the traditional format. These structure diagrams also provide a convenient framework within which to construct a formal derivation for an existing program.

We begin by giving a brief introduction to the refinement calculus, and showing how refinements are traditionally presented. We then show how annotated structure diagrams can be used to present refinements, and discuss some ways in which these diagrams can be made more readable. Next, we show how these structure diagrams can be used as a basis for constructing a derivation for an existing program. Finally, we discuss some related work and present our conclusions.

2 The Refinement Calculus

The refinement calculus (e.g. [Morgan, 1994], [Morgan and Vickers, 1994]) is a formalism for deriving imperative programs from logical specifications. Programs are derived in a wide-spectrum language based on Dijkstra's guarded commands language [Dijkstra, 1976], augmented with constructs for expressing specifications, which are (in general) not executable. Most notably, we can write a specification statement

$$w\colon \left[\, Pre \;\middle/\; Post \,\right]$$

which specifies a program that will make the postcondition *Post* true, altering only variables in w (known as the *frame*), provided that the precondition *Pre* is true initially.

For example, the following statement specifies a program to set m to the smallest element in a non-empty array segment $A[1\,..\,n]$:

$$m\colon \left[\, n \geq 1 \;\middle/\; m = \mathsf{min}\, A[1\,..\,n] \,\right]$$

Programs are constructed by performing transformations that preserve the refinement relation between programs. We say S is refined by S', written $S \sqsubseteq S'$, if every correct implementation of S' is also a correct implementation of S. This is defined formally in terms of weakest preconditions [Dijkstra, 1976] as:

$$S \sqsubseteq S' \;\text{ iff }\; \text{for every postcondition } R,\; wp(S, R) \;\Rightarrow\; wp(S', R)$$

A derivation consists of a sequence of programs in the wide spectrum language, where each program is a refinement of its predecessor. Each step usually transforms only a component of its predecessor. The final program is then a refinement of the original, because of transitivity and monotonicity properties of refinement ([Back, 1988], [Morris, 1987]).

To avoid having to reason about weakest preconditions directly, we normally justify refinement steps using *refinement rules* which embody laws stating that certain refinements are always valid, perhaps subject to certain applicability conditions or proof obligations. A small selection of refinement rules is shown below; a more extensive list of refinement laws can be found in [Morgan, 1994]. We write refinement rules as inference rules with the proof obligations (if any) above the line, and the inferred refinement below.[1]

Local Variable:

$$\frac{x \text{ does not occur in } w,\ P \text{ or } R}{w\colon\left[P \,\middle/\, R\right] \sqsubseteq [\![\ \mathbf{var}\ \ x \bullet x, w\colon\left[P \,\middle/\, R\right]]\!]}$$

Split Specification:

$$w\colon\left[P \,\middle/\, R\right] \sqsubseteq w\colon\left[P \,\middle/\, Q\right] ;\ w\colon\left[Q \,\middle/\, R\right]$$

Assignment:

$$\frac{P \ \Rightarrow\ R[x \leftarrow e]}{x, w\colon\left[P \,\middle/\, R\right] \ \sqsubseteq\ x := e}$$

where $R[x \leftarrow e]$ is R with all free occurrences of x replaced by e.

Leading Assignment:

$$\frac{x \text{ does not occur in } e \text{ or } P}{x, w\colon\left[P \,\middle/\, R\right] \ \sqsubseteq\ x := e\ ;\ w\colon\left[P \wedge x = e \,\middle/\, R\right]}$$

Following Assignment:

$$x, w\colon\left[P \,\middle/\, R\right] \sqsubseteq w\colon\left[P \,\middle/\, R[x \leftarrow e]\right] ;\ x := e$$

Alternation:

$$\frac{P \ \Rightarrow\ G_1 \ \vee\ G_2}{w\colon\left[P \,\middle/\, R\right] \ \sqsubseteq \begin{array}{ll} \mathbf{if} & G_1 \rightarrow w\colon\left[I \wedge G_1 \,\middle/\, R\right] \\ [\!] & G_2 \rightarrow w\colon\left[I \wedge G_2 \,\middle/\, R\right] \\ \mathbf{fi} & \end{array}}$$

Iteration:

$$\frac{\begin{array}{c} P \ \Rightarrow\ I \\ I \wedge \neg G \ \Rightarrow\ R \end{array}}{w\colon\left[P \,\middle/\, R\right] \ \sqsubseteq\ \mathbf{do}\ \ G \rightarrow w\colon\left[I \wedge G \,\middle/\, I \wedge 0 \leq t < t_0\right]\ \mathbf{od}}$$

where t_0 is t with each variable x in w replaced by x_0 (x_0 refers to the value of x prior to execution of the loop). I is the loop invariant and t is the variant function.

These rules are actually schemas, describing (usually infinite) sets of theorems about refinement. To apply a refinement rule whose bottom line is *before* $\sqsubseteq$ *after* to a program component *prog*, we instantiate the schema variables in the rule so that the resulting instance of *before* matches *prog*, and prove the corresponding instances of the applicability conditions. Values for schema variables that do not occur in *before* may be chosen freely, so long as the applicability conditions hold.

[1] Many of these rules can be generalised; for example, in their more general form, *Local Variable* can introduce several variables, *Assignment* can introduce a multiple assignment, and *Alternation* and *Iteration* can introduce an **if** or **do** statement with any number of guarded commands.

The rules shown above all operate on a specification statement and introduce some component of program structure. This is the kind of rule used most often in typical derivations. There are also rules that remove or transform program structure (e.g. deleting **skip** statements, or factoring out common parts of **if** or **do** statements), and data refinement rules, which transform a program operating on an abstract data type to one operating on a concrete representation of that type.

Example The following is a derivation of a program to find the smallest element in a non-empty array segment, presented in the style used in most program refinement literature (e.g. [Morgan, 1994]). Specification statements are labelled in the right-hand margin, and these labels are used to indicate the component being refined at each step. Each refinement step also shows the refinement rules used.[2]

We begin with the given specification:

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

We will need a loop with a counter, so we introduce a local variable for the counter.

$$\begin{array}{lll} (i) & \sqsubseteq & (\textit{Local Variable}) \\ & & \mathbf{var}\ k \bullet \\ & & m, k\colon \left[n \geq 1 \;\middle/\; m = \mathsf{min}\, A[1\,..\,n] \right] \qquad (ii) \end{array}$$

Next, we split the specification into two parts. One part will initialise the loop counter and establish the loop invariant; the other will become the loop. We use the loop invariant, $1 \leq k \leq n \wedge m = \mathsf{min}\, A[1\,..\,k]$, as the intermediate assertion.

$$\begin{array}{lll} (ii) & \sqsubseteq & (\textit{Split Specification}) \\ & & m, k\colon \left[n \geq 1 \;\middle/\; 1 \leq k \leq n \wedge m = \mathsf{min}\, A[1\,..\,k] \right]; \qquad (iii) \\ & & m, k\colon \left[1 \leq k \leq n \wedge m = \mathsf{min}\, A[1\,..\,k] \;\middle/\; m = \mathsf{min}\, A[1\,..\,n] \right] \qquad (iv) \end{array}$$

We can establish the loop invariant by setting k to 1 and m to $A[1]$, since $n \geq 1 \;\Rightarrow\; 1 \leq 1 \leq n \wedge A[1] = \mathsf{min}\, A[1\,..\,1]$ holds.

$$\begin{array}{lll} (iii) & \sqsubseteq & (\textit{Assignment}) \\ & & m, k := A[1], 1 \end{array}$$

We now introduce the loop, noting that $1 \leq k \leq n \wedge m = \mathsf{min}\, A[1\,..\,k] \wedge k = n \;\Rightarrow\; m = \mathsf{min}\, A[1\,..\,n]$ holds (the other proof obligation is trivial), and simplifying the precondition in the specification for the loop body.

[2] The format used here is not quite the same as Morgan's. Instead of labelling every specification statement, Morgan uses a simple convention to determine which specification is being refined when none is explicitly indicated.

$$(iv) \quad \sqsubseteq \quad (\textit{Iteration})$$

$$\begin{array}{l}\textbf{do } k \neq n \rightarrow \\ \quad m, k \colon \left[\begin{array}{c} 1 \leq k < n \\ m = \mathsf{min}\, A[1\,..\,k] \end{array} \middle/ \begin{array}{c} 1 \leq k \leq n \\ m = \mathsf{min}\, A[1\,..\,k] \\ 0 \leq k_0 < k \end{array} \right] \qquad (v) \\ \textbf{od}\end{array}$$

We split the loop body into two parts: one to increment k, which ensures that the loop makes progress towards termination, and one to re-establish the loop invariant. This is done using the *Following Assignment* rule, which constructs the intermediate assertion from the assignment statement. We can delete the terms involving k from the postcondition of the resulting specification, since they are implied by the precondition and this statement cannot change k (because our version of *Following Assignment* removes the variable assigned to in the second statement from the frame of the first).

$$(v) \quad \sqsubseteq \quad (\textit{Following Assignment})$$

$$\begin{array}{l} m \colon \left[1 \leq k < n \wedge m = \mathsf{min}\, A[1\,..\,k] \;\middle/\; m = \mathsf{min}\, A[1\,..\,k+1] \right]; \qquad (vi) \\ k := k + 1 \end{array}$$

In order to re-establish the loop invariant, we need to consider two cases, so we use an **if** statement; noting that $1 \leq k < n \wedge m = \mathsf{min}\, A[1\,..\,k] \;\Rightarrow\; m \leq A[k+1] \vee m \geq A[k+1]$.

$$(vi) \quad \sqsubseteq \quad (\textit{Alternation})$$

$$\begin{array}{l}\textbf{if } \; m \leq A[k+1] \rightarrow \\ \quad m \colon \left[\begin{array}{c} 1 \leq k < n \\ m = \mathsf{min}\, A[1\,..\,k] \\ m \leq A[k+1] \end{array} \middle/ \; m = \mathsf{min}\, A[1\,..\,k+1] \right] \qquad (vii) \\ [\!] \; m \geq A[k+1] \rightarrow \\ \quad m \colon \left[\begin{array}{c} 1 \leq k < n \\ m = \mathsf{min}\, A[1\,..\,k] \\ m \geq A[k+1] \end{array} \middle/ \; m = \mathsf{min}\, A[1\,..\,k+1] \right] \qquad (viii) \\ \textbf{fi}\end{array}$$

The first branch can be implemented as a **skip**, since $1 \leq k \leq n \wedge m = \mathsf{min}\, A[1\,..\,k] \wedge k \neq n \wedge m \leq A[k+1] \;\Rightarrow\; 1 \leq k+1 \leq n \wedge m = \mathsf{min}\, A[1\,..\,k+1]$ holds.

$$(vii) \quad \sqsubseteq \quad (\textit{Skip})$$

skip

The second branch can be implemented by setting m to $A[k+1]$, since $1 \leq k \leq n \wedge m = \mathsf{min}\, A[1\,..\,k] \wedge k \neq n \wedge m \geq A[k+1] \;\Rightarrow\; 1 \leq k+1 \leq n \wedge A[k+1] = \mathsf{min}\, A[1\,..\,k+1]$ holds.

$(viii) \quad \sqsubseteq \quad (\mathit{Assignment})$
$$m := A[k+1]$$

This completes the derivation. The resulting program is shown in Figure 1.

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

Figure 1: Program for minimum of array segment.

3 Refinement Structure Diagrams

The format illustrated in the preceding example provides a comprehensive record of the steps in the derivation, and their proof obligations. It is quite suitable for presenting derivations of intricate algorithms, allowing formal description of refinement steps to be interspersed with informal text explaining the design decisions made. This format is not, however, very convenient as a basis for other programming tasks.

Considering how a program derivation might be used in a broader range of programming tasks leads to a number of requirements. We want to be able to quickly see the status of an incomplete derivation, to determine what still needs to be done. We want to be able to see how decisions made during the derivation are related to the structure of the resulting program (for example, in understanding the performance properties of the program), and how this might be affected by different decisions. We also want to be able to determine what parts of the derivation will need to be changed to reflect a change in the specification, and to identify common or similar structures that might be factored out into procedures or tactics for reuse. Finally, we want a representation that will allow formally derived program fragments to be combined easily, and allow derivations to be constructed for existing programs (see Section 5).

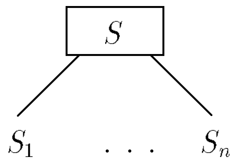
We propose to address these requirements using a form of structure diagram, augmented with formal specifications and related annotations. These diagrams, which we call Refinement Structure Diagrams, present the derivation in a familiar hierarchical fashion, reflecting the structure of the resulting program.

Structure diagrams have been used for many years as a diagrammatic representation for program designs. Many variations have been used by different authors (e.g. [Doran and Tate, 1972], [Jackson, 1975]), but typically, programs are represented as trees, where leaf nodes correspond to atomic actions, such as assignments, I/O operations or procedures calls, and non-leaf nodes correspond to control structures with their children as components. Non-leaf nodes also usually have informal text describing the action performed by the subtree rooted at that node, which may be included as comments in the final program.

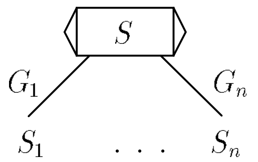
Our Refinement Structure Diagrams are based on a form of structure diagram where the non-leaf nodes correspond to the control structures in Dijkstra's guarded commands language. In particular, decision boxes (corresponding to **if**...**fi**) and repetition boxes (corresponding to **do**...**do**) may have an arbitrary number of children, which are interpreted as guarded commands.

In order to represent a formal derivation, we augment these structure diagrams, by adding formal specifications to the nodes, so that non-leaf nodes correspond to applications of refinement rules. We also adopt a number of other conventions that allow these diagrams to be displayed more compactly.

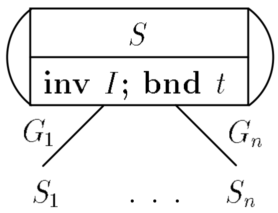
The following list shows the main structures used in constructing Refinement Structure Diagrams, along with the refinement represented and the refinement rule(s) used.



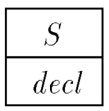
$S \sqsubseteq S_1;\ \cdots;\ S_n$, using (possibly repeatedly) *Sequential Composition* or more specialised rules such as *Leading Assignment* or *Following Assignment*.



$S \sqsubseteq \textbf{if}\ \ G_1 \rightarrow S_1\ [\!]\ \cdots\ [\!]\ G_n \rightarrow S_n\ \textbf{fi}$, using *Alternation*.

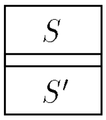


$S \sqsubseteq \textbf{do}\ \ G_1 \rightarrow S_1\ [\!]\ \cdots\ [\!]\ G_n \rightarrow S_n\ \textbf{od}$, using *Iteration*, with I as the loop invariant and t as the variant (or bound) function.



$S \sqsubseteq [\![\ \mathit{decl} \bullet S']\!]$, using one or more declaration introduction rules (e.g. *Local Variable*, *Local Definition*, *Local Invariant* or *Logical Constant*), where *decl* is one or more declarations and S' depends on the kinds of declarations introduced.[3]

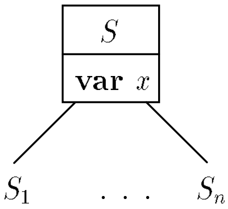
[3] For *Local Variable*, S' is the same as S with the declared variable(s) added to the frame; in other cases, S and S' are identical.



$S \sqsubseteq S'$, using a rule such as *Assignment*, *Skip*, *Abort* or *Procedure Call*, when S' is an atomic statement, or a rule such as *Weaken Precondition*, *Strengthen Postcondition* or *Contract Frame*, when S' is a specification statement.

To simplify structure diagrams, the latter rules may also be used silently in conjunction with the previous forms. Where two or more of these rules are applied in succession, they may be collapsed into a single node. The $S \sqsubseteq S'$ form may also be used when S' is an arbitrary program component, as a way of making the diagram more compact.

Note that declarations may be used in conjunction with the other forms, and modify their meanings in the (hopefully) obvious ways. For example, the form:



represents the refinement $S \sqsubseteq [\![\ \mathbf{var}\ x \bullet S_1;\ \cdots;\ S_n]\!]$, using *Local Variable* followed by *Sequential Composition*.

We can further annotate nodes with the names of refinement rules used, but in most cases this is unnecessary since the rules are usually obvious. A refinement tool based on Refinement Structure Diagrams would record the rules used, along with the proof obligations and their proofs, and explanatory text, and allow these to be selectively displayed.

Using the structures shown above, we can represent applications of rules that introduce program structure or transform specifications, which are the ones most commonly used in derivations. We can represent the application of rules that remove declarations, such as *Remove Logical Constant*, by obscuring the declaration in some way, e.g. by displaying it in a smaller font or using a lighter colour or shading. Rules that modify program structure, such as *Flatten Nested Alternations* or *Remove False Guard*,[4] or perform global transformations, such as data refinement, need to be regarded as transformations on structure diagrams.

Example The derivation given in Section 2 can be presented as a Refinement Structure Diagram, as shown in Figure 2.

This format is clearly more compact that the previous version. It is easier to see how different parts of the derivation are related, for example, where a given specification was introduced into the derivation and how it, in turn, is implemented. It is also easier to determine how the derivation would need to be changed to reflect a change in the specification; e.g. to find the maximum element in $A[1\,..\,n]$ instead of,

[4]These rules can be found in [Morgan, 1994].

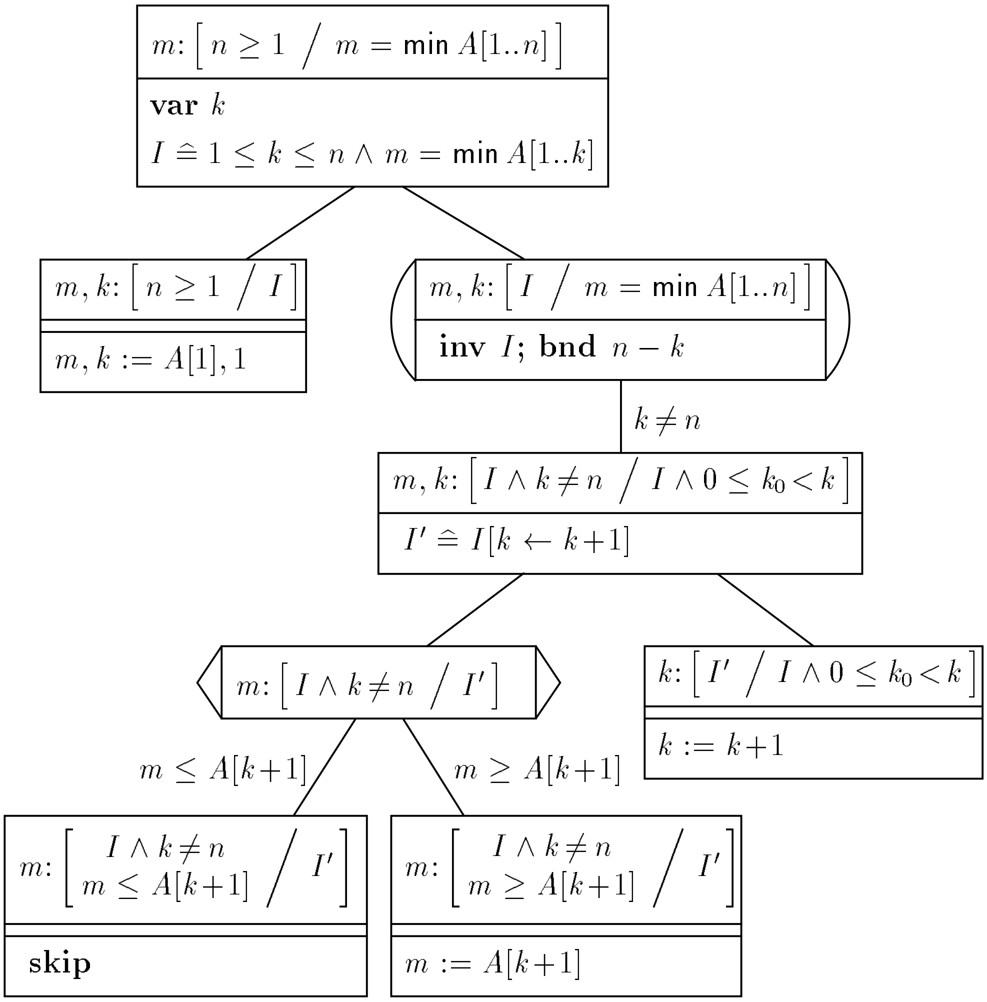


Figure 2: Refinement Structure Diagram for minimum of array segment.

or perhaps as well as, the minimum. Notice also that this format is not dependent on the precise order in which refinement steps are performed — this diagram would also represent a derivation in which the loop was introduced before the initialisation.

4 Procedures and Modularity

As Refinement Structure Diagrams become large, we need to be able to break them into several smaller diagrams, just as we break up a program into smaller parts using procedures, macros or modules. We allow the root of a structure diagram to be named, and allow the refinement in a leaf node to refer to a named structure diagram. This can be combined with Morgan's substitution notation [Morgan, 1988], or with a more conventional parameterisation mechanism.

In constructing the final program, we may choose to code separate structure diagrams as procedures, so the root of a named diagram corresponds to a procedure declaration and a reference to a diagram name is coded as a procedure call. Alternatively, we can regard separate diagrams simply as a convenient way of breaking up the description of the program design, and expand "calls" to other structure diagrams in-line in the final program. In a refinement tool, this choice could be made by annotating the root of a named structure diagram in some way, and this could be changed at any time. This means that the structure used to present the design need not correspond to the way that the final program is packaged into procedures, and can be chosen for convenience of understanding. See [Groves, 1996] for further discussion of this approach to procedures.

Example Suppose we want to make m the minimum of three numbers, x, y and z. We might implement this by first making m the minimum of x and y, and then making m the minimum of m and z. We can present this derivation as a Refinement Structure Diagrams as shown in Figure 3.

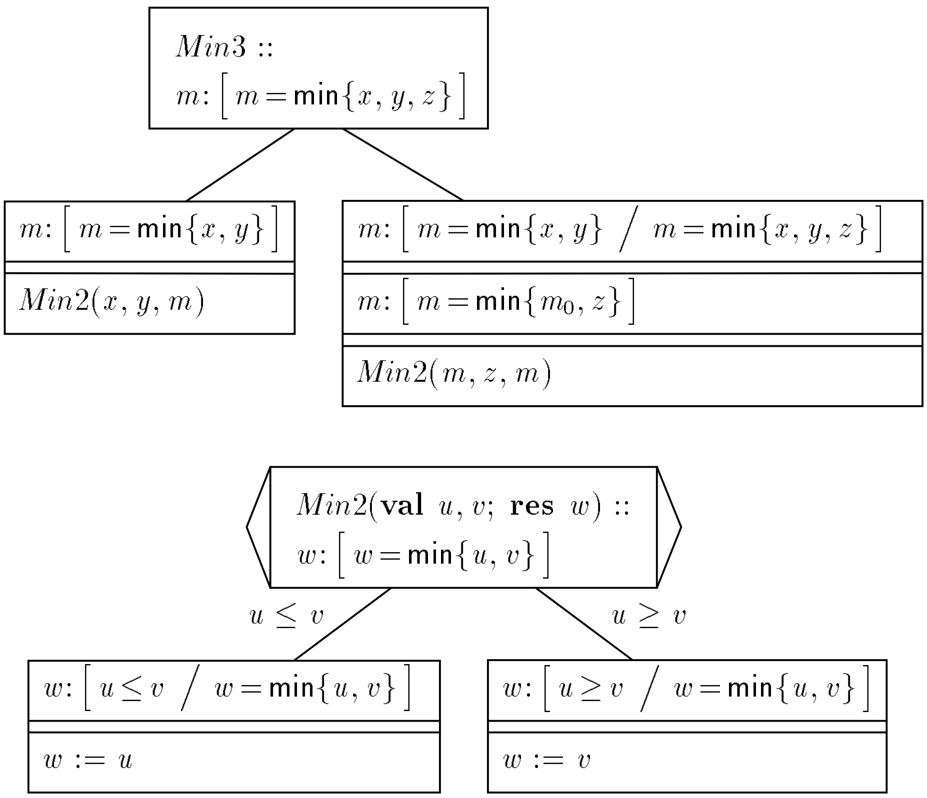


Figure 3: Refinement Structure Diagram using subdiagrams.

Note that this derivation uses an abbreviated form of specification statement in which the precondition is omitted. In this case, the omitted precondition is *true*.

Depending on whether we choose to code $Min2$ as a procedure, we could obtain either of the programs shown in Figure 4 from this derivation.

The refinement in a leaf node may refer to root of the same structure diagram, in which case the diagram must be coded as a recursive procedure. A refinement

$$
\begin{array}{l}
[\![\ \textbf{proc}\ Min2(\textbf{value}\ u, v;\ \textbf{result}\ w) \mathrel{\widehat{=}} \\
\qquad \textbf{if}\quad u \le v \to w := u \\
\qquad [\!]\quad u \ge v \to w := v \\
\qquad \textbf{fi} \bullet \\
\quad Min2(x, y, m); \\
\quad Min2(m, z, m) \\
]\!]
\end{array}
\qquad\qquad
\begin{array}{l}
\textbf{if}\quad x \le y \to m := x \\
[\!]\quad x \ge y \to m := y \\
\textbf{fi}; \\
\textbf{if}\quad m \le z \to m := m \\
[\!]\quad m \ge z \to m := z \\
\textbf{fi}
\end{array}
$$

Figure 4: Two programs for $Min3$.

tool based on this approach would check for such cycles and ask for a variant to be provided so that termination can be ensured.

5 Constructing a Derivation for an Existing Program

We will now show how Refinement Structure Diagrams can be used as a basis for constructing a formal derivation for an existing program. We can use essentially the same approach whether or not we have a specification for the program.

In this case, much of the structure of the derivation is determined by the structure of the program. To complete the derivation, we need to select appropriate rules, provide intermediate assertions, loop invariants and variant functions, and prove the resulting proof obligations.

We begin by constructing a plain structure diagram for the program. We then add specifications to the various nodes, incorporating as much information as we can from the form of the program, and from the initial specification if this is known, leaving any unknown formulas as metavariables [Nickson and Groves, 1994]. We collect constraints that these metavariables must satisfy, and substitute values for the metavariables when we have enough constraints to choose values for them.

Example Consider the following program:

$$
\begin{array}{l}
[\![\ \textbf{var}\ k \bullet \\
\quad k, r := 1, n + 1; \\
\quad \textbf{do}\ k \ne r \to \\
\qquad \textbf{if}\quad b(i) \to r := k \\
\qquad [\!]\quad \neg b(i) \to k := k + 1 \\
\qquad \textbf{fi} \\
\quad \textbf{od} \\
]\!]
\end{array}
$$

This program implements a bounded linear search algorithm (this is a variant of the algorithm derived in [Dijkstra and Feijen, 1989]). The algorithm is intended to set r to the smallest number $k \in 1..n$ for which $b(k)$ holds, or to $n+1$ if $b(k)$ is false for all $k \in 1..n$. So a reasonable specification (assuming that n can be any natural number) is:

$$r{:}\left[\, true \;\big/\; r = \mathsf{min}\ (\ \{\ i \ \mid\ 1 \le i \le n \wedge b(i)\ \} \cup \{n+1\}\ )\,\right]$$

In this case we have a specification, but it is not immediately obvious that the program does correctly implement this specification. We might thus wish to construct a derivation for the program in order to show that the program does indeed satisfy this specification.

Constructing a plain structure diagram for the program gives the structure diagram shown in Figure 5.

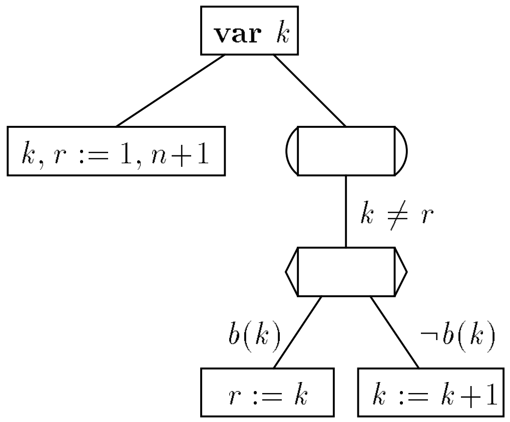


Figure 5: Plain structure diagram for search program.

We now add the given specification to the root node, and add specifications to the other nodes as dictated by the obvious refinement rules, leaving metavariables in places of unknown formulas. For brevity we abbreviate the postcondition to $r = MIN$. The resulting diagram is shown in Figure 6.

Most of the derivation is determined by the structure of the program, and the only unknowns are the loop invariant I and variant function t. Values for these must be selected so as to make the following proof obligations provable. The numbers beside nodes in the above diagram indicate the refinements to which they apply.

(1) $\quad true \;\Rightarrow\; I[k, r \leftarrow 1, n+1]$

(2) $\quad I \wedge i = k \;\Rightarrow\; k = MIN$

(3) $\quad I \wedge i \ne k \wedge b(k) \;\Rightarrow\; I[r \leftarrow k] \wedge 0 \le t[r \leftarrow k] < t_0$

(4) $\quad I \wedge i \ne k \wedge \neg b(k) \;\Rightarrow\; I[k \leftarrow k+1] \wedge 0 \le t[k \leftarrow k+1] < t_0$

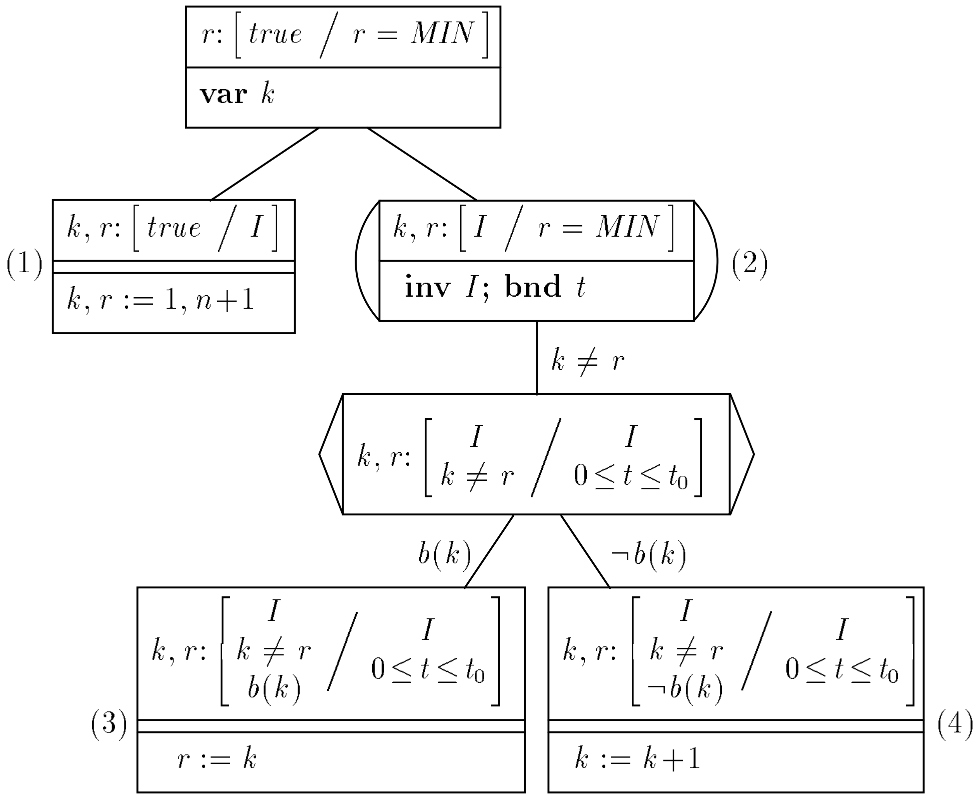


Figure 6: Refinement Structure Diagram with meta-variables.

It is easy to see that $r - k$ is initially non-negative, and is decreased by both branches of the **if** statement, so we can take $t = r - k$ as the variant function. The second conjunct of obligations (3) and (4) are now provable, provided that $k \le r$ holds.

With a bit of further investigation, we realise that k is at least 1 and at most $n + 1$, and that r is always that smallest value, i, such that either i is between 1 and $k - 1$ (inclusive) and $b(i)$ holds, or $i = n + 1$. Thus, we get the loop invariant:

$$I \mathrel{\widehat{=}} 1 \le k \le n + 1 \wedge r = \mathsf{min}\,\{\, i \mid (1 \le i \le k - 1 \wedge b(i)) \vee i = n + 1 \,\}$$

With this choice for I, it is straightforward to show that the above proof obligations hold, and that $k \le r$ holds. Thus, the program does satisfy our chosen specification.

Using the approach outlined above, we can construct a derivation for any program, using only the structure introducing rules listed in Section 2. This much could have been done using Hoare logic [Hoare, 1969]. Using the refinement calculus, we can also use rules that transform specifications (*Weaken Precondition*, *Strengthen Postcondition* and *Contract Frame*), which sometimes simplify the proof obligations, making the resulting derivation easier. We can also use rules that transform program structure or perform data refinement, allowing us to effectively restructure the program so as to simplify the derivation and the resulting proof obligations. This will normally require additional steps to be inserted into the structure diagram constructed from the program.

6 Conclusions

We have described an approach to presenting formal program derivations, based on structure diagrams annotated with formal specifications and details of refinement steps. This format is more compact than the traditional format used in most refinement calculus literature. Our experience suggests that this format makes it easier to understand the relationship between refinement steps and the final program, thus making it easier to see how changes in the specification might impact the derivation or how different design decisions might affect the program.

Other authors have proposed graphical representations of program derivations which share some features with ours. In Back's refinement diagrams [Back, 1991], program fragments are shown in textual form. Boxes are drawn around transformed components, with lines linking these to the result of the transformation. This representation is more general than ours, but is less compact. Naftalin's refinement trees [Naftalin, 1994] are more similar to our diagrams, though again they are less compact.

We are developing a refinement tool to support the use of Refinement Structure Diagrams. The formal annotations are optional and can be added at any stage in the development, so that the tool can support a variety of development styles. At one extreme, a development may be fully formal, in which case specifications are provided and proof obligations discharged as the diagram is constructed. At the other extreme, a development may be completely informal, in which case specifications and proofs are omitted completely and the tool acts as an ordinary structure diagram editor. Between these extremes formal and informal development may be combined in various ways, including constructing a formal derivation for an existing program, as illustrated in Section 5. We are also using Refinement Structure Diagrams as a basis for investigating techniques for modifying formal derivations to reflect specification changes, and for reusing formally derived program fragments.

We have defined the operations that may be performed on Refinement Stsructure Diagrams, and in particular the ways in which diagram nodes may be collapsed and combined, more formally than is presented here, and will describe this in a later paper. Further investigation is required to find good ways of portraying the effect of refinement rules that transform program structure, since these need to be treated as transformations on structure diagrams.

References

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

[Back, 1991] R. J. R. Back. Refinement diagrams. Proc. 4th Refinement Workshop, Springer Verlag, 1991, pp125–137.

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

[Dijkstra and Feijen, 1989] E. W. Dijkstra and W. H. J. Feijen. "The Linear Search Revisited". *Structured Programming* **10**, 1 (1989), pp5–9.

[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.

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

[Hoare, 1969] C. A. R. Hoare. "An axiomatic basis for computer programming". *C.A.C.M.* **12** (1969), pp576–580, 583.

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

[Morgan, 1988] Carroll Morgan. "Procedures, parameters, and abstraction: separate concerns". *Sci. Comp. Prog.* **11** (1988). Also in [Morgan and Vickers, 1994], pp58–71.

[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.

[Morris, 1987] J. M. Morris. "A theoretical basis for stepwise refinement and the programming calculus". *Science of Computer Programming* **9** (1987), pp287–306.

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

[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.