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 Refinement with Structure Diagrams

Lindsay Groves

Technical Report CS-TR-96/5
July 1996

Abstract

We present an approach to formal program derivation integrating ideas from the refinement calculus with structure diagrams, and discuss how it might be supported by a refinement tool. This approach allows program derivations to be presented, and constructed, in a diagrammatic formalism which is similar to traditional structure diagrams but has well-defined semantics. The approach allows formal and informal derivation to be combined, and allows formal derivations to be constructed in more flexible ways than existing refinement tools.

Publishing Information

Fifth Australasian Refinement Workshop, Brisbane, April 1996.

Program Refinement with Structure Diagrams

Lindsay Groves

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

Abstract

We present an approach to formal program derivation integrating ideas from the refinement calculus with structure diagrams, and discuss how it might be supported by a refinement tool. This approach allows program derivations to be presented, and constructed, in a diagrammatic formalism which is similar to traditional structure diagrams but has well-defined semantics. The approach allows formal and informal derivation to be combined, and allows formal derivations to be constructed in more flexible ways than existing refinement tools.

1 Introduction

In order for formal methods to become more widely accepted, we need to provide clear links with the more familiar informal methods and smooth paths for the gradual introduction of formal methods. This means being able to support a range from completely informal to completely formal development within a common framework. It is unrealistic at this stage to expect that all, or even most, software will be developed in a completely formal way, and there is some evidence to suggest that there are positive benefits in partly formal development.

Some work has been done on combining formal and informal methods for software specification (e.g. [Semmens, France and Docker, 1992]), but little has been done so for in the area of software development, a notable exception being the work of Maurice Naftalin [Naftalin, 1994]. In addition to being able to mix formal and informal development, we want to allow formal development to be done in a flexible manner. For example, as well as being able to do the development in a strictly formal top-down fashion, proving the correctness of each refinement step, we should also be able to build all or part of the program informally and later fill in the details of formal specifications, invariants etc. and prove the refinements.

In this paper, we present an approach based on integrating ideas from the refinement calculus with structure diagrams. This allows program development to be done in terms of a diagrammatic representation in which formal and informal approaches can be combined freely. Using hypertext techniques, a refinement tool can allow a user to choose to view a program development as a traditional (informal) structure diagram, as a formal development, or as a mixture of the two.

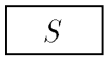
We begin by introducing our version of structure diagrams (Section 2) and showing how we augment these to include formal specifications (Section 3). We next explain how these augmented structure diagrams represent program derivations in the refinement calculus (Sections 4, 5 and 6) and how the approach supports a flexible style of formal development (Section 7). We then discuss the kind of tool support we envisage (Section 8) and present our conclusions (Section 9).

2 Structure Diagrams

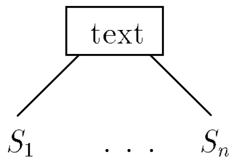
Structure diagrams have been used for many years as a diagrammatic representation for program designs (e.g. [Doran and Tate, 1972], [Jackson, 1975]). Many variations have been used, but typically, programs are represented as trees, where leaf nodes correspond to atomic actions, such as assignments, I/O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

We use a form of structure diagram where the non-leaf nodes correspond to the control structures in Dijkstra's guarded commands language [Dijkstra, 1976]. Thus, we have the following structures:

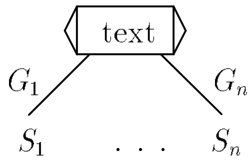
- For atomic statement S (assignment, skip, abort or procedure call) :



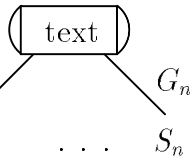
- For $S_1; \cdots; S_n$:



- For **if** $G_1 \rightarrow S_1 \,[\!]\, \cdots \,[\!]\, G_n \rightarrow S_n$ **fi** :

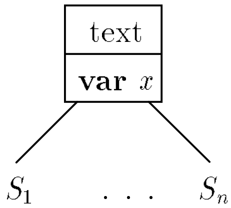


- For **do** $G_1 \rightarrow S_1 \,[\!]\, \cdots \,[\!]\, G_n \rightarrow S_n$ **od** :



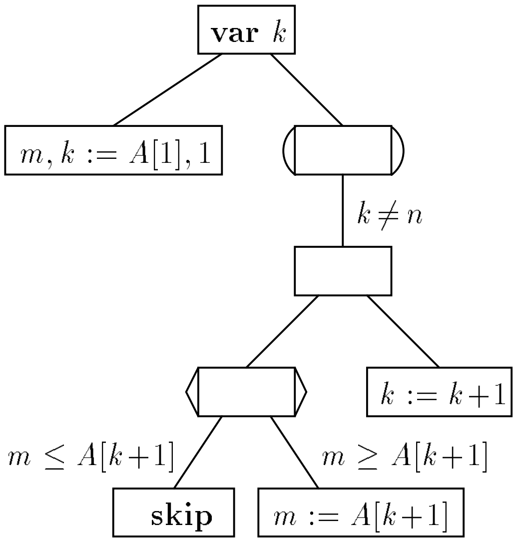
We also write $\boxed{\text{text}}$ for a non-expanded node in a partially completed structure diagram, and change the shape of the box if necessary when the node is expanded.

To avoid cluttering diagrams with additional nodes and edges just to introduce declarations, such as local variables, we add such declarations to the node at the root of the subtree which is their scope. For example, we use



for $[\![$ **var** $x \bullet S_1;\ \cdots;\ S_n]\!]$.

Example The following is a structure diagram representing a program to set m to the minimum value in a non-empty array segment $A[1..n]$:



This structure diagram only shows the essential structure, not the informal textual comments that would normally be included.

The corresponding program in Dijkstra's guarded commands language is:

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

Structure diagrams are intended to be used in constructing programs in a top-down fashion. The informal text associated with nodes plays an essentially role in the development. This text provides a specification for the program component represented by the subtree rooted at that node. Thus, that subtree is to be constructed in such a way that the resulting program component correctly implements this specification. Of course, since the specification is expressed informally, the notion of correctness is also informal, relying on an informal reasoning process known as "inspection".

Note that as well as describing the structure of a program, a structure diagram provides a record (albeit informal) of the reasoning process by which it was (or might have been) constructed. The textual comments express the programmer's intentions at each step.

3 Adding Formal Specifications

In order to provide a more formal notion of the correctness of a program represented by a structure diagram, and a more formal record of its development, we need to replace (or supplement) the informal specifications associated with its nodes by formal specifications. We will use specification statements, as used in the refinement calculus [Morgan, 1994]. In the case of nodes representing loops, we also attach loop invariants and variant (or bound) functions.

A specification statement $w\colon \left[\mathit{pre} \;\middle/\; \mathit{post} \right]$ will establish the postcondition *post*, changing only variables in the frame w, provided that the precondition *pre* holds initially. For example, the program to set m to the minimum value in a non-empty array segment $A[1..n]$ is specified by:

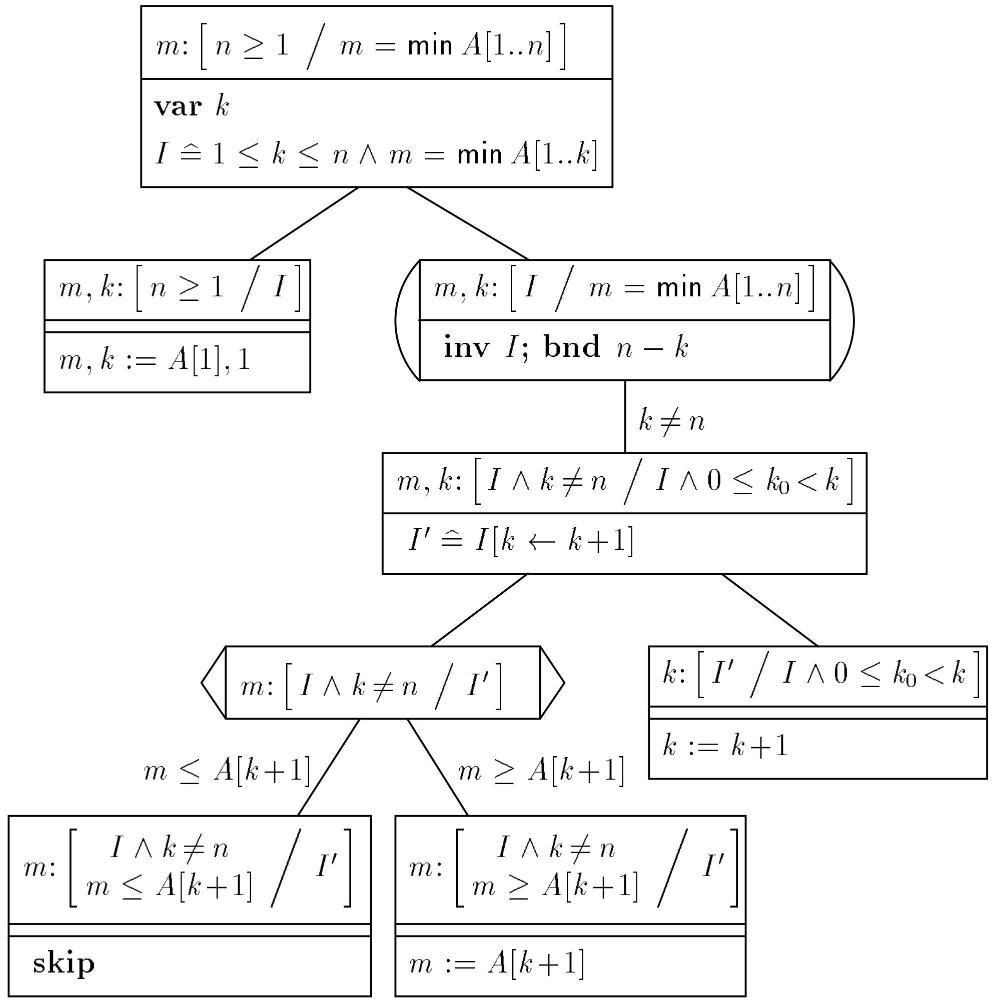
$$m\colon \left[n \geq 1 \;\middle/\; m = \mathsf{min}\, A[1 \mathinner{..} n] \right]$$

Note that the specification only allows m to be modified, so A and n cannot be changed. Since the program is only required to return a sensible result when the precondition, $n \geq 1$, holds, it can do anything at all (including not terminate) if the precondition does not hold.

We adopt some further notation from the refinement calculus to help make structure diagrams more readable: we write $x \mathrel{\widehat{=}} y$ for syntactic definitions, so x can be used as an abbreviation for y, and we write $x[v := e]$ for the expression obtained by replacing all free occurrences of v in x by e.

When specification statement gets large, we may use an alternative layout, with the precondition above the postcondition, separated by a horizontal line. We might ultimately want to find a way of displaying specifications that is more visually appealing, but for now we are content to stay close to the usual refinement calculus notation.

Example The following is a version of the previous structure diagram for finding the minimum in an array, augmented with specifications:



By adding specifications in this way, we are able to associate a well-defined meaning with partially completed structure diagrams. They represent programs in the wide-spectrum language obtained by adding specification statements to Dijkstra's guarded commands language.

4 Structure Diagrams and Refinement

In the refinement calculus ([Morgan, 1994], [Morgan and Vickers, 1994]), the refinement relation $\sqsubseteq$ is defined such that $S \sqsubseteq S'$ (read "S is refined by S'") if S' can be used anywhere that S is required. When S is a specification and S' an executable program, this means that S' is a correct implementation of S. Since refinement is transitive and the program constructors are monotonic with respect to refinement, we can transform a specification into a program which implements that specification by successively replacing non-executable components (typically specification statements) by refinements which are closer to code. These transformations are usually justified by refinement rules which embody theorems stating that certain refinements are always valid, subject to the validity of certain proof obligations. The most common refinement rules turn specification statements into compound statements or atomic statements, or introduce local declarations.

We can understand a structure diagram augmented with specifications, as described above, as describing a program derivation in the refinement calculus. Each node corresponds to the application of one or more refinement rules, and is a valid refinement if the corresponding proof obligations hold.

Example The following is the refinement represented by the previous structure diagram, as it would normally be presented. 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.[1]

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

$(i) \quad \sqsubseteq \quad$ (*Introduce Local Variable*)

$$\begin{array}{l} \mathbf{var}\ k \bullet \\ m, k\colon \left[n \geq 1 \;\middle/\; m = \mathsf{min}\, A[1 \mathinner{..} n] \right] \qquad (ii) \end{array}$$

$(ii) \quad \sqsubseteq \quad$ (*Split Specification*)

$$\begin{array}{l} m, k\colon \left[n \geq 1 \;\middle/\; 1 \leq k \leq n \wedge m = \mathsf{min}\, A[1 \mathinner{..} k] \right]; \qquad (iii) \\ m, k\colon \left[1 \leq k \leq n \wedge m = \mathsf{min}\, A[1 \mathinner{..} k] \;\middle/\; m = \mathsf{min}\, A[1 \mathinner{..} n] \right] \qquad (iv) \end{array}$$

$(iii) \quad \sqsubseteq \quad$ (*Introduce Assignment*)

$$m, k := A[1], 1$$

$(iv) \quad \sqsubseteq \quad$ (*Introduce DO*)

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

[1] The refinement rules referred to can mostly be found in [Morgan, 1994].

$$
\begin{array}{lll}
(v) & \sqsubseteq & (\textit{Following Assignment}) \\
 & & m\colon \left[1 \le k < n \wedge m = \mathsf{min}\, A[1 \mathinner{..} k] \;\middle/\; m = \mathsf{min}\, A[1 \mathinner{..} k+1] \right]; \qquad (vi) \\
 & & k := k+1
\end{array}
$$

$$
\begin{array}{lll}
(vi) & \sqsubseteq & (\textit{Introduce IF}) \\
 & & \mathbf{if} \quad m \le A[k+1] \rightarrow \\
 & & \quad m\colon \left[\begin{array}{c} 1 \le k < n \\ m = \mathsf{min}\, A[1 \mathinner{..} k] \\ m \le A[k+1] \end{array} \middle/ \; m = \mathsf{min}\, A[1 \mathinner{..} k+1] \right] \qquad (vii) \\
 & & [\!] \quad m \ge A[k+1] \rightarrow \\
 & & \quad m\colon \left[\begin{array}{c} 1 \le k < n \\ m = \mathsf{min}\, A[1 \mathinner{..} k] \\ m \ge A[k+1] \end{array} \middle/ \; m = \mathsf{min}\, A[1 \mathinner{..} k+1] \right] \qquad (viii) \\
 & & \mathbf{fi}
\end{array}
$$

$$
\begin{array}{lll}
(vii) & \sqsubseteq & (\textit{Introduce Skip}) \\
 & & \mathbf{skip}
\end{array}
$$

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

The validity of the above derivation, and thus the correctness of the resulting program, depends on the validity of the following proof obligations, which are easily proved:

(iii) $n \ge 1 \Rightarrow 1 \le 1 \le n \wedge A[1] = \mathsf{min}\, A[1 \mathinner{..} 1]$

(iv) $1 \le k \le n \wedge m = \mathsf{min}\, A[1 \mathinner{..} k] \wedge k = n \Rightarrow m = \mathsf{min}\, A[1 \mathinner{..} n]$

(iv) $1 \le k \le n \wedge m = \mathsf{min}\, A[1 \mathinner{..} k] \wedge k \ne n \Rightarrow m \le A[k+1] \vee m \ge A[k+1]$

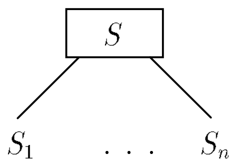
(vii) $1 \le k \le n \wedge m = \mathsf{min}\, A[1 \mathinner{..} k] \wedge k \ne n \wedge m \le A[k+1] \Rightarrow 1 \le k+1 \le n \wedge m = \mathsf{min}\, A[1 \mathinner{..} k+1]$

$(viii)$ $1 \le k \le n \wedge m = \mathsf{min}\, A[1 \mathinner{..} k] \wedge k \ne n \wedge m \ge A[k+1] \Rightarrow$
$1 \le k+1 \le n \wedge A[k+1] = \mathsf{min}\, A[1 \mathinner{..} k+1]$

5 Representing Derivations

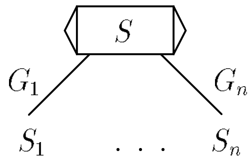
We will now explain more precisely how our augmented structure diagrams correspond to derivations in the refinement calculus.

- The form:



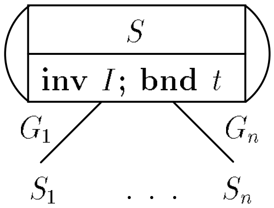
represents the refinement $S \sqsubseteq S_1; \cdots; S_n$, and corresponds to the (possibly repeated) application of *Sequential Composition* or more specialised rules such as *Leading Assignment* or *Following Assignment*.

- The form:



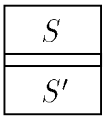
represents the refinement $S \sqsubseteq$ **if** $G_1 \to S_1 \; [] \cdots [] \; G_n \to S_n$ **fi**, and corresponds to the application of *Introduce If*.

- The form:



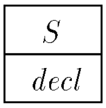
represents the refinement $S \sqsubseteq$ **do** $G_1 \to S_1 \; [] \cdots [] \; G_n \to S_n$ **od**, and corresponds to the application of *Introduce Do*, using I as the loop invariant and t as the variant (or bound) function.

- The form:

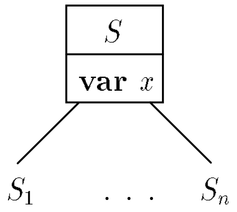


represents the refinement $S \sqsubseteq S'$. This is used when S' is an atomic statement, in which case it corresponds to the application of a rule such as *Introduce Assignment*, *Introduce Skip*, *Introduce Abort* or *Procedure Call*. It is also used when S' is a specification, in which case it corresponds to the application of a rule such as *Weaken Precondition*, *Strengthen Postcondition* or *Contract Frame*. 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 form:



indicates the introduction of one or more local declarations (local variable, local definition, local invariant or logical constant), using the appropriate rule(s). Note that the presence of declarations modifies the meanings of the previous forms; for example, the form:



represents the refinement $S \sqsubseteq [\![\textbf{var}\ x \bullet S_1;\ \cdots;\ S_n]\!]$, and corresponds to the application of *Introduce Local Variable* followed by an application of *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 would record the rules used and also the proof obligations and their proofs, and allow these to be selectively displayed.

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

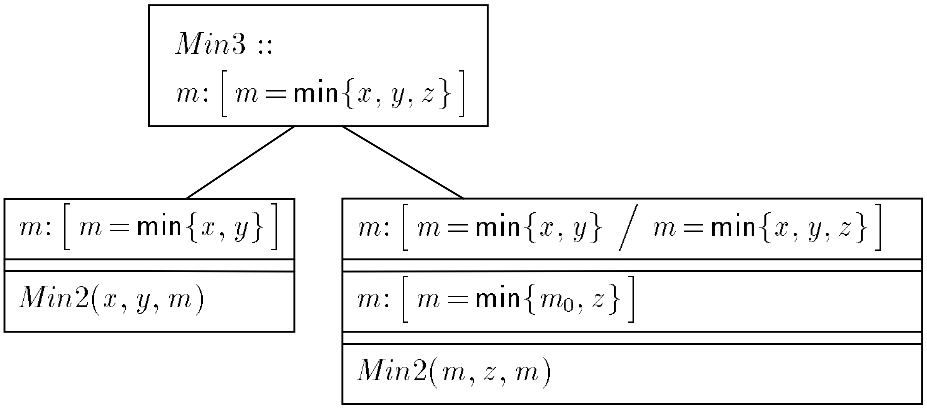
6 Procedures and Modularity

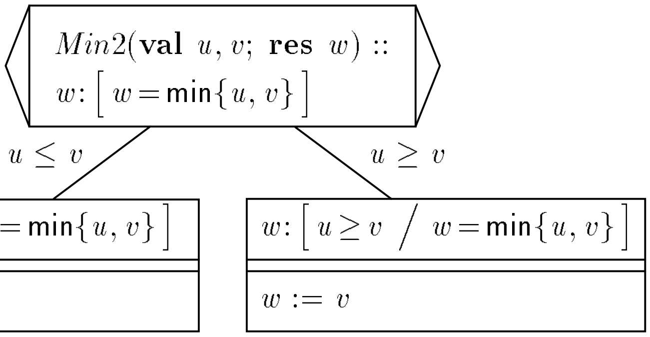
As structure diagrams become large, we will want to be able to break them into several smaller diagrams, just as we break up a program into smaller parts using procedures, macros or modules. To facilitate this, we need to allow the root of a structure diagram to be named and allow the refinement in a leaf node refer to another structure diagram by its name.[2] This can be combined with Morgan's substitution notation [Morgan, 1988], or with a more conventional parameterisation mechanism.

We can regard separate structure diagrams as procedures, and code them using procedure declarations and calls in the final program. Or we can regard them 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. 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 in terms of structure diagrams as follows:

[2] In fact, we could allow any node of a structure diagram to be named and "called" in this way.





The program resulting from this derivation could be either:

$\llbracket$ **proc** $Min2($**value** u, v; **result** $w) \mathrel{\widehat{=}}$
  if $u \le v \to w := u \;[\!]\; u \ge v \to w := v$ **fi** $\bullet$
 $Min2(x, y, m)$;
 $Min2(m, z, m)$
$\rrbracket$

or:

if $x \le y \to m := x \;[\!]\; x \ge y \to m := y$ **fi**;
if $m \le z \to m := m \;[\!]\; m \ge z \to m := z$ **fi**

according to whether or not $Min2$ was marked as being a procedure.

We can also allow the refinement in a leaf node refer to root of the same structure diagram, in which case the corresponding procedure is recursive. A refinement tool based on this approach would check for such cycles and ask for a variant to be provided so that termination can be ensured.

7 Mixing Formal and Informal Development

We now consider how we can mix formal and informal development with our approach, and illustrate a particular pattern of development which we believe would prove useful in practice.

We often want to construct a formal derivation for an existing algorithm: perhaps to illustrate or experiment with refinement techniques or a refinement tool, or to help understand why the algorithm works. In this case, much of the structure of the derivation is determined by the structure of the algorithm. What is needed is to select appropriate rules, provide intermediate assertions, loop invariants and variant functions, and prove the resulting proof obligations.

Similarly, where the algorithm is not known, we may wish to sketch out part of the algorithm, e.g. to get an idea of its structure and computational properties, before working through the derivation formally.

With our approach, we can construct a plain structure diagram for the algorithm (or part of it). We can then add the initial specification (assuming that it is know[3]) and specifications for the various nodes. In doing this, we leave any unknown formulas as metavariables [Nickson and Groves, 1994] and collect constraints that they must satisfy, and substitute values for the metavariables when we have enough constraints to choose values for them.

Example Consider the following bounded linear search algorithm (this is a variant of the algorithm derived in [Dijkstra and Feijen, 1989]):

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

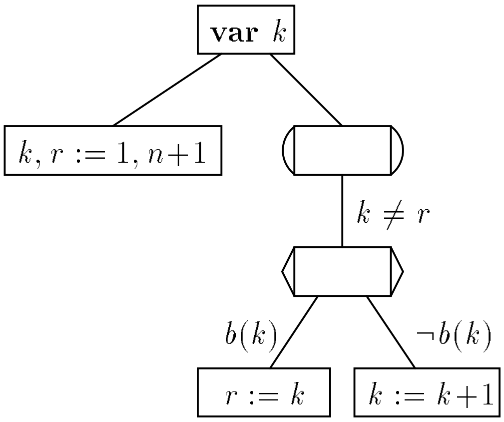
The algorithm is intended to set r to the smallest number $k \in 1..n$ for which $b(k)$ holds, if b ever holds within this range, and $n+1$ if $b(k)$ is false for all $k \in 1..n$. So a reasonable specification (assuming that n can be any natural) is:

$$r\colon \left[\, \mathit{true} \ \middle/\ r = \mathsf{min}\ (\{\, i \mid 1 \leq i \leq n \wedge b(i)\,\} \cup \{n+1\}) \,\right]$$

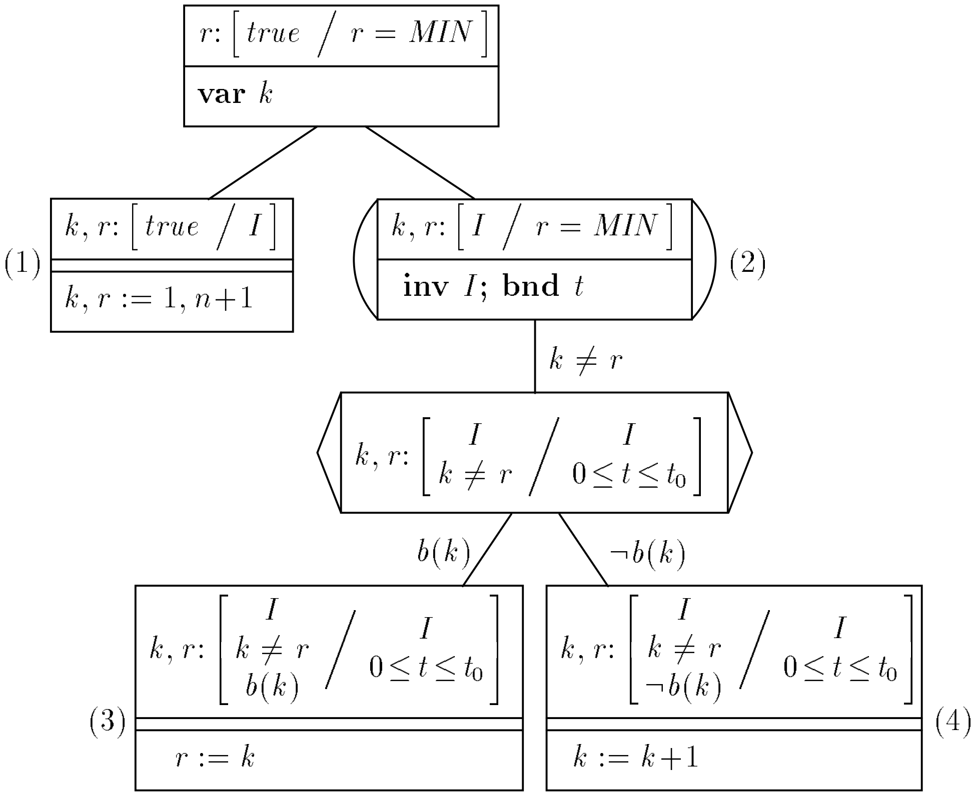
In order to show that the algorithm does indeed satisfy this specification, we would like to construct a derivation for the algorithm from the specification. Most of the derivation is determined by the structure of the algorithm — the important step is finding the loop invariant.

We can begin by constructing a plain structure diagram for the algorithm:

[3] If the specification is not known, we might use these techniques to help discover it.



We can then add the specification to the initial node, and add partial specifications[4] to the other nodes as dictated by the obvious refinement rules. For brevity we abbreviate the postcondition to $r = MIN$.



All we need now is to find suitable values for the loop invariant I and variant function t. 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.

[4]These specifications are partial in the sense that some of the expressions required have not yet been determined.

(1) $I[k, r := 1, n+1]$

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

(3) $I \wedge i \neq k \wedge b(k) \Rightarrow I[r := k] \wedge 0 \leq t[r := k] < t_0$

(4) $I \wedge i \neq k \wedge \neg b(k) \Rightarrow I[k := k+1] \wedge 0 \leq t[k := k+1] < t_0$

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 $k \leq r$ 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 $(1 \leq i \leq k-1 \wedge b(i)) \vee i = n+1$. Thus, we get the loop invariant:

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

With this choice for I, it is straightforward to show that the above proof obligations hold, and that $k \leq r$ holds.

8 Tool Support

It is widely recognised that successful use of formal methods requires tool support to handle routine tasks such as formula manipulation, applying transformation rules and recording proof obligations. Being based on a diagrammatic formalism, the approach presented here lends itself well to interactive tool support. Such a refinement tool would provide many of the facilities normally found in existing editors for diagrammatic notations and in existing refinement tools (e.g. see [Groves, Nickson and Utting, 1992], [Nickson, 1994]). We now outline some of the facilities we envisage being provided in such a tool which are peculiar to this approach.

A refinement tool based on our approach would allow a structure diagram to built in an informal way, just as with existing editors for such diagrams. The tool would also allow specifications and other formal annotations, such as loop invariants, refinement rules and proof obligations to be associated with nodes. These formal annotations could be provided as the structure diagram is being constructed, thus supporting a fully formal derivation style, or added later. The nodes in a structure diagram could be marked in some way (e.g. using different colours) to indicate whether they had formal specifications associated with them, whether the refinements they represent had been justified using appropriate refinement rules, and whether the corresponding proof obligations had been discharged.

When displaying a structure diagram, the user should be able to display either just the informal specifications, just the formal specifications, or both. This could be controlled either by global settings or by settings on individual nodes.

9 Conclusions

We have presented an approach to constructing programs which allows both formal program derivation and informal program construction to be combined within a single framework. By combining structure diagrams with ideas from the refinement calculus, we can associate well-defined semantics with augmented structure diagrams, and understand a structure diagram as representing a derivation in the same sense as the refinement calculus.

The approach is well suited to interactive tool support, and can allow the formal parts of a derivation to be constructed in more flexible ways than existing refinement tools.

References

[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, Nickson and Utting, 1992] Lindsay Groves, Raymond Nickson and Mark Utting. "A Tactic Driven Refinement Tool". *Proc. 5th Refinement Workshop*, Springer-Verlag, 1992, pp272–297.

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

[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.

[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.

[Semmens, France and Docker, 1992] L.T. Semmens, R.B. France and T.W.G. Docker, "Integrated Structured Analysis and Formal Specification Techniques". *The Computer Journal* **35**, 6 (1992), pp600–610.