Concurrent Program Design in the Extended Theory of Owicki and Gries

Doug Goldson **Brijesh Dongol**[†]

School of ITEE
University of Queensland,
Brisbane, Australia
Email: [†]brijesh@itee.uq.edu.au

Abstract

Feijen and van Gasteren have shown how to use the theory of Owicki and Gries to design concurrent programs, however, the lack of a formal theory of progress has meant that these designs are driven entirely by safety requirements. Proof of progress requirements are made post-hoc to the derivation and are operational in nature. In this paper, we describe the use of an extended theory of Owicki and Gries in concurrent program design. The extended theory incorporates a logic of progress, which provides opportunity to develop a program in a manner that gives proper consideration to progress requirements. Dekker's algorithm for two process mutual exclusion is chosen to illustrate the use of the extended theory.

1 Introduction

Concurrent programs are delicate entities and are consequently difficult to get right. Inherent difficulties in conventional testing exist, which have long been recognised. Unfortunately, formal verification is also a daunting task, made more difficult by the fact that program code cannot be altered. Complex programs invariably give rise to complex proofs, and when things become complicated, it is hard to judge whether the program or the proof strategy is at fault. For this reason, we have chosen to explore an approach to concurrent program design that is based on derivation. A program and its proof are developed hand in hand and carefully judged modifications are made to a program. Modifications are driven by an attention to proof obligations derived from the initial requirements.

The theory of Owicki and Gries (Owicki & Gries 1976, Dijkstra 1982) has long been a popular choice for verifying concurrent programs, and more recently (Feijen & van Gasteren 1999) describes how the theory can be used for program derivation. However, the original theory is limited by the fact that it lacks a formal theory of progress. This means that the verificaton or derivation of a concurrent program can only address safety requirements, with progress requirements being largely left to chance, or reasoned about operationally. In (Dongol & Goldson 2004) we describe how the theory of Owicki and Gries can be extended with a logic of progress, a development that provides an opportunity to drive program derivation in a manner that gives proper consideration to both progress and safety requirements. The main purpose of this paper is to show how the extended theory can be used to derive programs in this way.

We have chosen to illustrate the extended theory with a derivation of Dekker's algorithm for two process mutual exclusion. We have chosen this program because mutual exclusion is a core problem in concurrent programming, because Dekker's algorithm is the first successful solution to this problem, and, most of all, because nearly all of the program code is concerned with ensuring progress.

The paper is structured as follows. Section 2 reviews the extended theory of Owicki and Gries. Section 3 presents Dekker's algorithm and sets the context for our derivation of it, which is presented in Section 4. Section 5 makes a conclusion.

2 The extended theory of Owicki and Gries

This section reviews essential background to the main part of the paper which is the derivation presented in Section 4. It is intended to make the paper self-contained, but the reader should note that full explanation can be found in (Dongol & Goldson 2004).

2.1 The programming model

We begin by reviewing the programming language and model used in the derivation, and the terminology used in this paper. We call a *sequential program* a *component*, which is just a program statement. We call a *concurrent program* simply a *program*, and this is a collection of components, together with a precondition that defines its initial states.

A program statement is a statement in the language of guarded commands (Dijkstra 1976).

Definition A *statement* S is defined by,

$$\begin{array}{rl} S ::= & \mathit{skip} \mid \overline{x{:=}\,E} \mid S_0;\ S_1 \mid \langle S \rangle \\ & \mid \textbf{if } B_0 \to S_0 \,[]\, \ldots \,[]\, B_m \to S_m \textbf{ fi} \\ & \mid \textbf{do } B_0 \to S_0 \,[]\, \ldots \,[]\, B_m \to S_m \textbf{ od} \end{array}$$

□

Here, $\overline{x{:=}\,E} \mathrel{\widehat{=}} x_1{:=}\,E_1 \parallel \ldots \parallel x_n{:=}\,E_n$ is a multiple assignment, $S_0;\ S_1$ is the sequential composition of statements S_0 and S_1, $\langle S \rangle$ is a coarse-grained atomic statement, which is any statement S enclosed in atomicity brackets $\langle\ \rangle$, an **if** statement is used to express choice (and condition synchronisation), and a **do** statement is used to express repetition.

The atomic statements of the programming language are *skip*, the assignment statement, and the guard evaluation statement, which, it may be noted, can only appear in a program as a part of an **if** or **do** statement. An atomic statement must be enabled for it to be executed, and when it is executed it results in a single update of the control state of the program (which means that it is guaranteed to terminate when

it is executed). A guard evaluation statement in an **if** statement is a conditional atomic statement, in the sense that it is not enabled when all of the guards in the statement are evaluated false. In this case, execution of the guard evaluation statement is blocked, and this makes the **if** statement a well-suited mechanism for programming condition synchronisation between components. We allow one more atomic statement in the programming language, which is the coarse-grained statement $\langle S \rangle$. Being atomic, execution of this statement must terminate and so it is only enabled when termination is guaranteed. The programming model prescribes that the underlying machine is weakly fair, which means that on termination of an atomic statement, an atomic statement that follows it, if there is one, is eventually executed if it is continually enabled. This means that in the concurrent execution of a number of components, the execution of the next (continually enabled) atomic statement of no component is delayed indefinitely.

2.2 Representing program control points

In order to reason about progress, it is necessary to introduce a means to describe the control points in a program. In the extended theory of Owicki and Gries, this is done in two steps. First every atomic statement in a component is assigned a label unique to that component. Second, a program counter is introduced into each component.

We illustrate the use of labels by labelling a program called the safe sluice (Figure 1), which plays an important role in the derivation to come.

Pre: $\neg x \wedge \neg y$	
Component X	Component Y
0: **do** $true \rightarrow$	0: **do** $true \rightarrow$
1: $\langle NCS_X \rangle$;	1: $\langle NCS_Y \rangle$;
2: $x := true$;	2: $y := true$;
3: **if** $\neg y \rightarrow$	3: **if** $\neg x \rightarrow$
4: $skip$	4: $skip$
fi;	**fi**;
5: $\langle CS_X \rangle$;	5: $\langle CS_Y \rangle$;
6: $x := false$	6: $y := false$
od	**od**

Figure 1: Safe sluice with labels

The essential property of a component labelling is that the atomic statements of a component are in one-to-one correspondence with the labels of the component. For this reason, we use the notation $A.i$, where i is not the final label of A, to designate 'the atomic statement in component A labelled i'. In this way, for example, $X.3$ is used to pick out the guard evaluation statement of the **if** statement in component X above. Note that the correspondence between atomic statements and labels is achieved in this example by making the code segments *NCS* and *CS* into coarse-grained atomic actions. Anticipating our use of the safe sluice in deriving a mutual exlusion protocol, *NCS* and *CS* designate non-critical and critical sections of code. By assuming the internals of this code to be not relevant to the safe sluice, we are free to turn *NCS* and *CS* into atomic statements.

The next step is to introduce program counters into the two components X and Y. A program counter is modelled by an auxiliary variable (Figure 2).

The essential property of a program counter is that, for any component A and label i, $pc_A = i$ is a correct program assertion at label i, where 'correct' means correct in the core theory of Owicki and Gries presented in Section 2.3. And for this reason we are free to interpret assertion $pc_A = i$, for i not the final

Pre: $\neg x \wedge \neg y \wedge pc_X = pc_Y = 0$	
Component X	Component Y
0: **do** $\langle true \rightarrow pc_X := 1 \rangle$	0: **do** $\langle true \rightarrow pc_Y := 1 \rangle$
1: $\langle NCS_X;\ pc_X := 2 \rangle$;	1: $\langle NCS_Y;\ pc_Y := 2 \rangle$;
2: $x := true \parallel pc_X := 3$;	2: $y := true \parallel pc_Y := 3$;
3: **if** $\langle \neg y \rightarrow pc_X := 4 \rangle$	3: **if** $\langle \neg x \rightarrow pc_Y := 4 \rangle$
4: $\langle skip;\ pc_X := 5 \rangle$	4: $\langle skip;\ pc_Y := 5 \rangle$
fi;	**fi**;
5: $\langle CS_X;\ pc_X := 6 \rangle$;	5: $\langle CS_Y;\ pc_Y := 6 \rangle$;
6: $x := false \parallel pc_X := 0$	6: $y := false \parallel pc_Y := 0$
od	**od**

Figure 2: Safe sluice with program counters

label of A, to mean that 'program control in component A is at the atomic statement labelled i'.

2.3 Hoare logic, the wlp and the core theory of Owicki and Gries

A significant semantic benefit to our approach to modelling program control is that it allows us to retain the weakest liberal precondition *wlp* predicate transformer as a means to reason about the correctness of a component. The definition of *wlp* (Dijkstra 1976) is applied to a labelled statement with program counter *pc* as follows

1. $wlp.(i{:}\ \langle skip;\ pc{:=}\ j \rangle\ j{:}).P \equiv wlp.(pc{:=}\ j).P$
2. $wlp.(i{:}\ \overline{x{:=}\ E} \parallel pc{:=}\ j\ j{:}).P \equiv P[\overline{x{:=}\ E} \parallel pc{:=}\ j]$
3. $wlp.(i{:}\ \langle S;\ pc{:=}\ j \rangle\ j{:}).P \equiv wlp.(S;\ pc{:=}\ j).P$
4. $wlp.(i{:}\ S_0;\ j{:}\ S_1\ k{:}).P$
   $\equiv$
   $wlp.(i{:}\ S_0\ j{:}).(wlp.(j{:}\ S_1\ k{:}).P$
5. $wlp.(i{:}\ \mathbf{if}\ \langle B_0 \rightarrow pc{:=}\ j \rangle j{:}\ S_0$
   $[\!]\ \langle B_1 \rightarrow pc{:=}\ k \rangle k{:}\ S_1\ \mathbf{fi}\ l{:}).P$
   $\equiv$
   $(B_0 \Rightarrow wlp.(pc{:=}\ j).(wlp.(j{:}\ S_0\ l{:}).P)) \wedge$
   $(B_1 \Rightarrow wlp.(pc{:=}\ k).(wlp.(k{:}\ S_1\ l{:}).P))$
6. $\{P\}\ i{:}\mathbf{do}\ \langle B \rightarrow\ pc{:=}\ j \rangle\ j{:}\ S$
   $[\!]\ \langle \neg B \rightarrow pc{:=}\ k \rangle\ \mathbf{od}\ k{:}\ \{Q\}$
   $\Leftarrow$
   $((P \wedge B \Rightarrow wlp.(pc{:=}\ j).(wlp.(j{:}\ S\ i{:}).P)) \wedge$
   $(P \wedge \neg B \Rightarrow wlp.(pc{:=}\ k).Q)$

On the other hand, a significant syntactic drawback to our approach is that the code of a component is complicated by the superimposition of program counter assignments onto every atomic statement in the component. To avoid this problem we allow ourselves to leave these cluttering program counter assignments implicit in the component code, defining the *wlp* for a labelled statement with implicit program counter *pc* as follows

1. $wlp.(i{:}\ skip\ j{:}).P \equiv wlp.(pc{:=}\ j).P$
2. $wlp.(i{:}\ \overline{x{:=}\ E}\ j{:}).P \equiv P[\overline{x{:=}\ E} \parallel pc{:=}\ j]$
3. $wlp.(i{:}\ \langle S \rangle\ j{:}).P \equiv wlp.(S;\ pc{:=}\ j).P$
4. $wlp.(i{:}\ S_0;\ j{:}\ S_1 k{:}).P$
   $\equiv$
   $wlp.(i{:}\ S_0\ j{:}).(wlp.(j{:}\ S_1\ k{:}).P)$
5. $wlp.(i{:}\ \mathbf{if}\ B_0 \rightarrow\ j{:}\ S_0\ [\!]\ B_1 \rightarrow\ k{:}\ S_1\ \mathbf{fi}\ l{:}).P$
   $\equiv$
   $(B_0 \Rightarrow wlp.(pc{:=}\ j).(wlp.(j{:}\ S_0\ l{:}).P)) \wedge$
   $(B_1 \Rightarrow wlp.(pc{:=}\ k).(wlp.(k{:}\ S_1\ l{:}).P))$
6. $\{P\}\ i{:}\mathbf{do}\ B \rightarrow\ j{:}\ S\ \mathbf{od}\ k{:}\ \{Q\}$
   $\Leftarrow$
   $((P \wedge B \Rightarrow wlp.(pc{:=}\ j).(wlp.(j{:}\ S\ i{:}).P)) \wedge$
   $(P \wedge \neg B \Rightarrow wlp.(pc{:=}\ k).Q)$.

We can now use the predicate transformer *wlp* to calculate whether execution of a program statement satisfies a postcondition given that some precondition is assumed to be true when the statement is executed. In other words, we can use the *wlp* to reason about a *Hoare-triple* $\{P\}\ i{:}\ S\ j{:}\ \{Q\}$, given that the triple is true whenever $P \Rightarrow wlp.(i{:}\ S\ j{:}).Q$. We

also define $pre(i) \mathrel{\widehat{=}} P$ and $post(j) \mathrel{\widehat{=}} Q$. Hoare-triples are the logical basis of program annotation, which, in turn, is the logical basis of the core theory of Owicki and Gries.

Rule (*Local Correctness*) An assertion P in a component is *locally correct* (LC) when,

1. if P is textually preceded by program precondition Pre, then $Pre \Rightarrow P$
2. if P is textually preceded by $\{Q\}\ S$, then $\{Q\}\ S\ \{P\}$ holds.

Rule (*Global Correctness*) An assertion P in a component is *globally correct* (GC) if for each $\{Q\}\ S$ from a different component, $\{P \wedge Q\}\ S\ \{P\}$ holds.

To illustrate how the core theory of Owicki and Gries can be used to verify satisfaction of a safety requirement, we next present a proof that the safe sluice satisfies the following requirement.

Safety: "Components X and Y are not in their critical sections CS_X and CS_Y at the same time."

On account of already having made the code segments *NCS* and *CS* into atomic statements! we formalise this requirement by defining a *critical set* C of control points

$$C \mathrel{\widehat{=}} \{4, 5\}$$

that represents a component being about to execute its $\langle CS \rangle$ statement. While control point 4 is not at $\langle CS \rangle$, the fact that there is no blocking code between control point 4 and $\langle CS \rangle$ justifies its inclusion in the critical set C. Satisfaction of the safety requirement now amounts to proving the following invariant

$$I \mathrel{\widehat{=}} pc_X \in C \Rightarrow pc_Y \notin C.$$

Proof of safety. I is invariant if the annotation in Figure 3 is correct.

Pre: $\neg x \wedge \neg y \wedge pc_X = pc_Y = 0$ Invariant: $pc_X \in C \Rightarrow pc_Y \notin C$	
Component X	Component Y
0: **do** *true* $\rightarrow$	0: **do** *true* $\rightarrow$
1: $\langle NCS_X \rangle$;	1: $\langle NCS_Y \rangle$;
2: x:= *true*;	2: y:= *true*;
3: **if** $\neg y \rightarrow \{?\ pc_Y \notin C\}$	3: **if** $\neg x \rightarrow \{?\ pc_X \notin C\}$
4: *skip*	4: *skip*
fi;	**fi**;
5: $\langle CS_X \rangle$;	5: $\langle CS_Y \rangle$;
6: x:= *false*	6: y:= *false*
od	**od**

Figure 3: Annotated safe sluice

Local correctness (LC) of the assertions $pre(X.4)$ and $pre(Y.4)$ are achieved (in the only way possible) by investing in invariants

$$J_1 \mathrel{\widehat{=}} \neg y \Rightarrow pc_Y \notin C$$
$$J_2 \mathrel{\widehat{=}} \neg x \Rightarrow pc_X \notin C.$$

It is not difficult to see that these are indeed invariant. For J_1, y is a LC assertion at all control points in Y where $pc_Y \in C$ and a GC assertion by the fact that y is a private variable of Y (where private means not written to in any other component).

Now we verify GC of the assertions $pre(X.4)$ and $pre(Y.4)$ from Figure 3. We will need to establish GC of $pre(X.4)$, under statement $Y.3$ — the **if** guard evaluation statement in Y, and under statement $Y.4$, as they are both able to falsify $pc_Y \notin C$ by an implicit update of the program counter in Y. Against $Y.3$, $pc_Y \notin C$ is only GC in the presence of coassertion x, which ensures that execution of this guard evaluation statement is blocked. Fortunately, x is a correct coassertion of $pc_Y \notin C$ and we get the correct annotation in Figure 4 on account of

Pre: $\neg x \wedge \neg y \wedge pc_X = pc_Y = 0$ Invariants: I: $pc_X \in C \Rightarrow pc_Y \notin C$ J_1: $\neg y \Rightarrow pc_Y \notin C$ J_2: $\neg x \Rightarrow pc_X \notin C$	
Component X	Component Y
0: **do** *true* $\rightarrow$	0: **do** *true* $\rightarrow$
1: $\langle NCS_X \rangle$;	1: $\langle NCS_Y \rangle$;
2: x:= *true*;	2: y:= *true*;
3: **if** $\neg y \rightarrow \{pc_Y \notin C\}\{x\}$	3: **if** $\neg x \rightarrow \{pc_X \notin C\}\{y\}$
4: *skip*	4: *skip*
fi;	**fi**;
5: $\langle CS_X \rangle$;	5: $\langle CS_Y \rangle$;
6: x:= *false*	6: y:= *false*
od	**od**

Figure 4: Safe sluice correctly annotated

GC of $pre(X.4)$ under statement $Y.3$

$$pc_Y \notin C \wedge x \wedge \neg x \Rightarrow wlp.(pc_Y := 4).(pc_Y \notin C \wedge x)$$
$$\equiv$$
$$false \Rightarrow false$$

We get GC of $pre(X.4)$ under statement $Y.4$ for free by realizing that $pc_Y = 4$ is true at $Y.4$, which falsifies $pre(X.4)$, and the proof follows.

2.4 The extended theory of Owicki and Gries

The extended theory of Owicki and Gries is obtained from the core theory by the addition of a logic of progress. Any such logic is dependent upon the behaviour of the machine that executes a program, and the programming model described in Section 2.1 mandates that this machine satisfies a *weak fairness* assumption that an atomic statement is eventually executed whenever it is at an active control point and it is continually enabled. Assertions about progress in a program are formalised using a *leads-to* (denoted $\rightsquigarrow$) relation where $P \rightsquigarrow Q$ means that if P is true then Q is eventually true. Three rules define this relation, of which the most basic rule, which we call Immediate Progress, is defined in terms of another relation *unless* (**un**). We therefore first present the definition of *unless*.

Definition If P and Q are any two predicates, P **un** Q is true if the Hoare-triple

$$\{P \wedge \neg Q \wedge U\}\ S\ \{P \vee Q\}$$

is true for all atomic statements $\{U\}\ S$, where U denotes the precondition of S in the annotated program.

Relation *unless* says that a program state in which P holds and Q does not is perpetuated until a state is reached in which Q holds. But note that this does not guarantee that Q will ever hold, for (an extreme) example, *true* **un** Q holds for all Q, including *false*. The rules that define *leads-to* are now as follows.

Rule (*Immediate Progress Rule*) : $P \rightsquigarrow Q$ holds whenever there is a labelled statement with initial label i in a component with program counter pc and

1. P **un** Q
2. $P \wedge \neg Q \Rightarrow pc = i$
3. (a) The statement is an assignment or *skip* statement

   $i{:}\, S\ j{:}$ and,
   $P \wedge \neg Q \Rightarrow wlp.(i{:}\, S\ j{:}).Q$

   (b) The statement is an *IF* statement

   $i{:}\,\mathbf{if}\ B_0 \rightarrow\ j{:}\, S_0 \,[]\, B_1 \rightarrow\ k{:}\, S_1\ \mathbf{fi}\ l{:}$ and,
   (i) $P \wedge \neg Q \Rightarrow B_0 \vee B_1$
   (ii) $(P \wedge \neg Q \wedge B_0 \Rightarrow wlp.(pc{:=}\, j).Q)) \wedge$
   $(P \wedge \neg Q \wedge B_1 \Rightarrow wlp.(pc{:=}\, k).Q))$

   (c) The statement is a *DO* statement

   $i{:}\,\mathbf{do}\ B \rightarrow\ j{:}\, S\ \mathbf{od}\ k{:}$ and,
   $(P \wedge \neg Q \wedge B \Rightarrow wlp.(pc{:=}\, j).Q) \wedge$
   $(P \wedge \neg Q \wedge \neg B \Rightarrow wlp.(pc{:=}\, k).Q)$

   (d) The statement is a coarse-grained atomic statement

   $i{:}\,\langle S\rangle\ j{:}$ and,
   $P \wedge \neg Q \Rightarrow wp.S.(Q[pc{:=}\, j])$

Rule (*Inductive Progress Rules*)

(*Transitivity*) $P \rightsquigarrow R \Leftarrow P \rightsquigarrow Q \wedge Q \rightsquigarrow R$

(*Disjunction*) For any set W,
$(\exists\, i{:}\, i \in W{:}\, P_i) \rightsquigarrow Q \Leftarrow (\forall\, i{:}\, i \in W{:}\, P_i \rightsquigarrow Q)$

To make sense of these rules we provide these interpretative notes. First of all, we remark that this logic of progress is, in essence, that of UNITY (Chandy & Misra 1988, Misra 2001), but with some key changes made to the UNITY logic in the rule of immediate progress above. Our explanation of the rules will therefore focus on this one. The rule of transitivity requires no explanation. The rule of disjunction, in its finite application of, say, two progress assertions, amounts to the inference that if $P_0 \rightsquigarrow Q$ and $P_1 \rightsquigarrow Q$ then $P_0 \vee P_1 \rightsquigarrow Q$. In the case of immediate progress $P \rightsquigarrow Q$ is justified on the basis of being able to actually exhibit a continually enabled atomic statement at an active control point that makes Q true when it is executed. To see how the rule formalises this, we first note that $P \wedge \neg Q$ is assumed. Clause 1 of the rule establishes that P remains true as long as $\neg Q$ is true. Clause 2 establishes that control is at an atomic statement labelled i in a component. Clause 3 establishes that this statement is enabled when $P \wedge \neg Q$ is true, and that its execution makes Q true. It follows from clause 1 that the statement is continually enabled as long as $\neg Q$ is true. It then follows from the programming model that the statement is eventually executed. Clause 3 is separated into three cases to cover the three kinds of atomic statements. In case (3a) an assignment statement is always enabled and it is enough to ensure that its execution makes Q true. In case (3b) a guard evaluation statement in an **if** statement is not always enabled and so clause (3bi) ensures that it is enabled when $P \wedge \neg Q$ is true. Clause (3bii) further ensures that its execution makes Q true. In case (3c) a guard evaluation statement in a **do** statement is always enabled and it is again enough to ensure that its execution makes Q true.

3 Dekker's algorithm

This section presents Dekker's algorithm (Figure 5) for two process mutual exclusion. The algorithm as presented is a labelled version of that in (Feijen & van Gasteren 1999)(pp90). Dekker's algorithm was developed in the early 1960s and, historically, is the first solution to the mutual exclusion problem. We have chosen it as our example because we believe it is not possible to provide this program with a *convincing* operationally based argument that it satisfies its progress requirement. Given the intricacy of the code, nor are we optimistic of a verificationist approach to proving correctness of the program. For these reasons, Dek- ker's program represents a challenging exercise in program derivation, and therefore a good test of the extended theory of Owicki and Gries. There are two program requirements, one for safety and one for progress.

Safety: "Components X and Y are not in their critical sections CS_X and CS_Y at the same time."

Progress: "A component that is waiting to enter its CS code eventually does so."

Pre: $\neg x \wedge \neg y \wedge (v = X \vee v = Y)$

Component X	Component Y
0: **do** *true* →	0: **do** *true* →
1: $\langle NCS_X\rangle$;	1: $\langle NCS_Y\rangle$;
2: x:= *true*;	2: y:= *true*;
3: **if** $\neg y$ →	3: **if** $\neg x$ →
4: *skip*	4: *skip*
[] y →	[] x →
5: **if** $v = X$ →	5: **if** $v = Y$ →
6: *skip*	6: *skip*
[] $v \neq X$ →	[] $v \neq Y$ →
7: x:= *false*;	7: y:= *false*;
8: **if** $v = X$ →	8: **if** $v = X$ →
9: *skip*	9: *skip*
fi;	**fi**;
10: x:= *true*	10: y:= *true*
fi;	**fi**;
11: **if** $\neg y$ →	11: **if** $\neg x$ →
12: *skip*	12: *skip*
fi	**fi**
fi;	**fi**;
13: $\langle CS_X\rangle$;	13: $\langle CS_Y\rangle$;
14: v:= Y;	14: v:= X;
15: x:= *false*	15: y:= *false*
od	**od**

Figure 5: Dekker's algorithm

Some interpretive notes on the two requirements are in order. We apply the same interpretation to the safety requirement as was applied to the safe sluice in Section 2.3. Having made the code segments CS_X and CS_Y atomic (Figure 5), the safety requirement is formalised by defining a *critical set* of control points that guarantee eventual execution of statements $\langle CS_X\rangle$ and $\langle CS_Y\rangle$. However, as we do not propose to verify Dekker's algorithm, we do not need to define this set here. The progress requirement for, component X amounts to ensuring that X is never continually blocked at a synchronisation statement. Hence, eventual execution of $\langle CS_X\rangle$ when X is waiting means that when X is at $X.8$ eventually it is at $X.9$ and when X is at $X.11$ eventually it is at $X.12$.

At this point we invite the reader to convince themself that Dekker's algorithm satisfies the two requirements, and to pay particular attention to the progress requirement. In the words of (Feijen & van Gasteren 1999) "the argumentation should not be carried out superficially ... but carefully and meticu-

lously; then we gather that long before the argument is completed, the reader will see the light: this is like all hell let loose" (pp90-91).

Finally, we motivate the safe sluice of Section 2.3 as the starting point of our derivation because it has already been shown to satisfy the safety requirement. The progress requirement for the safe sluice, on the other hand, is another matter, with progress of X immediately doubtful on account of guard $\neg y$ in X oscillating in Y. Worse is the danger of total deadlock, as revealed by the possibility of correctly annotating X with x at statement $X.3$ and Y with y at statement $Y.3$. Further motivation is arrived at by careful examination of Dekker's algorithm at statements $X.2, 3, 11, 15$. These statements correspond to the statements $X.2, 3, 6$ that preserve safety in the safe sluice algorithm and highlight just how much of Dekker's algorithm is in the service of the progress requirement.

4 The derivation

This section describes a series of refinements that result in Dekker's algorithm. The pattern we will follow is that each refinement is motivated either by the safety requirement or the progress requirement. The development is necessarily delicate, as each refinement is at risk of violating a property of the program that has already been proved. Code modification is kept to a minimum, which is to say that each refinement step is kept small. The starting point of the derivation is the safe sluice program of Figure 4.

Refinement 1

PROOF (of progress at $X.3$) $pc_X = 3 \rightsquigarrow pc_X = 4$ involves showing that

(1) when the $X.3$ guard is *false*, it eventually becomes *true*, and

(2) when the $X.3$ guard is *true*, we eventually get past the guard.

This is formalised as

(1) $pc_X = 3 \wedge y \rightsquigarrow pc_X = 3 \wedge \neg y$

(2) $pc_X = 3 \wedge \neg y \rightsquigarrow pc_X = 4$

We have already remarked on the possibility of total deadlock in the safe sluice. Two design options now present themselves, we can either retain $X.3$ as a synchronisation statement, but weaken the guard $\neg y$ (so that the statement blocks in fewer program states). Alternatively, we could give up $X.3$ as a synchronisation statement. We choose the second option, but also decide to retain $\neg y$ as an entry condition for CS_X, which allows us to retain invariants J_1 and J_2 (cf Figure 6). While it is clear that a new synchronization statement and waiting condition is required when y is true, the exact choice is unclear at this stage, and so we defer the choice with template guard $B.X$ in the first refinement (Figure 6).

Recalling that critical set C represents a component about to execute its $\langle CS \rangle$ statement, the first refinement induces a redefinition of C to

$$C \mathrel{\widehat{=}} \{4, 5, 8\}.$$

Now, for the continued invariance of I we invest in two more invariants

$$K_1 \mathrel{\widehat{=}} B.X \Rightarrow pc_Y \notin C$$
$$K_2 \mathrel{\widehat{=}} B.Y \Rightarrow pc_X \notin C.$$

Pre: $\neg x \wedge \neg y \wedge pc_X = pc_Y = 0$
Invariants:
I: $pc_X \in C \Rightarrow pc_Y \notin C$ $\quad$ K_1: ? $B.X \Rightarrow pc_Y \notin C$
J_1: $\neg y \Rightarrow pc_Y \notin C$ $\quad$ K_2: ? $B.Y \Rightarrow pc_X \notin C$
J_2: $\neg x \Rightarrow pc_X \notin C$

Component X	Component Y
0: **do** *true* →	0: **do** *true* →
1: $\langle NCS_X \rangle$;	1: $\langle NCS_Y \rangle$;
2: x:= *true*;	2: y:= *true*;
3: **if** $\neg y \rightarrow \{pc_Y \notin C\}\{x\}$	3: **if** $\neg x \rightarrow \{?\ \neg B.X\}$
4: *skip*	4: *skip*
▯ $y \rightarrow$	▯ $x \rightarrow$
7: **if** $B.X \rightarrow$	7: **if** $B.Y \rightarrow \{?\ \neg B.X\}$
$\{?\ pc_Y \notin C\}$	8: *skip*
8: *skip*	**fi**
fi	**fi**;
fi;	5: $\langle CS_Y \rangle$;
5: $\langle CS_X \rangle$;	6: y:= *false*
6: x:= *false*	**od**
od	

Figure 6: Refinement 1 (i)

By the similarity of invariants K_i and J_i, it appears that we have come full circle, however, there is a crucial difference in that there is now choice for $B.X$. For the satisfaction of progress, we must show

(1) $pc_X = 7 \wedge \neg B.X \rightsquigarrow pc_X = 7 \wedge B.X$

(2) $pc_X = 7 \wedge B.X \rightsquigarrow pc_X = 8$.

In light of (1) and because $X.7$ is a synchronisation statement with guard $B.X$, we decide to set up component Y to make $B.X$ true, while for (2), we decide to set up component Y not to make $B.X$ false so that the potential problem of an oscillating guard can be avoided.

Now consider invariance of K_1

$$\begin{array}{l} B.X \Rightarrow pc_Y \notin C \\ \equiv pc_Y \in C \Rightarrow \neg B.X \end{array}$$

This requires $\neg B.X$ at both $Y.4$ and $Y.8$. We can solve $\neg B.X$ at $Y.8$ with $B.Y \Rightarrow \neg B.X$. Indeed, we will choose $B.Y \equiv \neg B.X$ to establish GC of $\neg B.X$ on account of having just decided that Y cannot falsify $B.X$ and so, by symmetry, nor can X falsify $B.Y$.

$\neg B.X$ at $Y.4$ is problematic. The obvious choice $\neg x \equiv \neg B.X$ is ruled out because it makes $\neg x \equiv B.Y$ and, by symmetry, $\neg y \equiv B.X$, but we have just decided that Y cannot falsify $B.X$. Instead we choose to weaken the annotation at $Y.4$, and drastically so, by weakening $\neg B.X$ to $\neg B.X \vee true$. This amounts to giving up on invariant K_1 and yields the annotation in Figure 7.

Summary: For reasons of progress, this refinement relocates the synchronization statement in the safe sluice from $X.3$ to $X.7$ and we are free to choose any guard $B.X$ such that $B.Y \equiv \neg B.X$ and $B.X$ is GC in X. A solution to this equation is to introduce a fresh variable $v \in \{X, Y\}$ such that

$$B.X \mathrel{\widehat{=}} v = X$$
$$B.Y \mathrel{\widehat{=}} v = Y.$$

Refinement 2

The proof obligations for progress have now migrated to the new synchronisation statement at $X.7$.

PROOF (of progress at $X.7$) $pc_X = 7 \rightsquigarrow pc_X = 8$ involves showing that

(1) $pc_X = 7 \wedge v \neq X \rightsquigarrow pc_X = 7 \wedge v = X$

Pre: $\neg x \wedge \neg y \wedge pc_X = pc_Y = 0$
Invariants:
I: $pc_X \in C \Rightarrow pc_Y \notin C$ $\quad B.Y \equiv \neg B.X$
J_1: $\neg y \Rightarrow pc_Y \notin C$
J_2: $\neg x \Rightarrow pc_X \notin C$

Component X	Component Y
0: **do** *true* $\rightarrow$	0: **do** *true* $\rightarrow$
1: $\langle NCS_X \rangle$;	1: $\langle NCS_Y \rangle$;
2: x:= *true*;	2: y:= *true*;
3: **if** $\neg y \rightarrow \{pc_Y \notin C\}\{x\}$	3: **if** $\neg x \rightarrow$
4: *skip*	4: *skip*
$[]\ y \rightarrow$	$[]\ x \rightarrow$
7: **if** $B.X \rightarrow \{B.X\}$	7: **if** $B.Y \rightarrow \{\neg B.X\}$
$\{?\ pc_Y \notin C\}$	8: *skip*
8: *skip*	**fi**
fi	**fi**;
fi;	5: $\langle CS_Y \rangle$;
5: $\langle CS_X \rangle$;	6: y:= *false*
6: x:= *false*	**od**
od	

Figure 7: Refinement 1 (ii)

(2) $pc_X = 7 \wedge v = X \rightsquigarrow pc_X = 8$.

And indeed we have made 'progress' in the derivation, because (2) now follows by the immediate progress rule on account of our earlier decision to make $v = X$ GC in X. For (1) it is clear that we need to make an assignment to v in Y that makes $v = X$ true, and in light of the role of $X.7$ as a synchronisation statement, the earliest safe opportunity to do this is at statement $Y.6$. The result is the annotation of Figure 8.

Pre: $\neg x \wedge \neg y \wedge (v = X \vee v = Y) \wedge pc_X = pc_Y = 0$
Invariants:
I: $pc_X \in C \Rightarrow pc_Y \notin C$
J_1: $\neg y \Rightarrow pc_Y \notin C$
J_2: $\neg x \Rightarrow pc_X \notin C$

Component X	Component Y
0: **do** *true* $\rightarrow$	0: **do** *true* $\rightarrow$
1: $\langle NCS_X \rangle$;	1: $\langle NCS_Y \rangle$;
2: x:= *true*;	2: y:= *true*;
3: **if** $\neg y \rightarrow \{pc_Y \notin C\}\{x\}$	3: **if** $\neg x \rightarrow$
4: *skip*	4: *skip*
$[]\ y \rightarrow \{x\}$	$[]\ x \rightarrow$
7: **if** $v = X \rightarrow \{v = X\}$	7: **if** $v = Y \rightarrow \{v = Y\}$
$\{?\ pc_Y \notin C\}$	8: *skip*
8: *skip*	**fi**
fi	**fi**;
fi;	5: $\langle CS_Y \rangle$;
5: $\langle CS_X \rangle$;	6: y:= *false* $\parallel$ v:= X
6: x:= *false* $\parallel$ v:= Y	**od**
od	

Figure 8: Refinement 2 (i)

It remains to check (1), which is done by considering all control points in Y. That is, we show

$$(\forall\, i :: pc_X = 7 \wedge v \neq X \wedge pc_Y = i \rightsquigarrow pc_X = 7 \wedge v = X).$$

Noting that $pc_X = 7$ is stable in Y, by immediate progress we get

$$\begin{array}{ll}
\quad v \neq X \wedge pc_Y = 0 & \\
\rightsquigarrow v \neq X \wedge pc_Y = 1 & \\
\rightsquigarrow v \neq X \wedge pc_Y = 2 & \\
\rightsquigarrow v \neq X \wedge pc_Y = 3 & \{pc_X = 7 \Rightarrow x\} \\
\rightsquigarrow v \neq X \wedge pc_Y = 7 & \{v \neq X \equiv v = Y\} \\
\rightsquigarrow v \neq X \wedge pc_Y = 8 & \\
\rightsquigarrow v \neq X \wedge pc_Y = 5 & \\
\rightsquigarrow v \neq X \wedge pc_Y = 6 & \\
\rightsquigarrow v = X &
\end{array}$$

$$\begin{array}{ll}
\quad v \neq X \wedge pc_Y = 4 & \\
\rightsquigarrow v \neq X \wedge pc_Y = 5 &
\end{array}$$

This gives us a proof of progress, which relies on the assumption that code segments CS and NCS will always terminate. However, it is usual to remove this assumption in the case of NCS. In terms of our formalisation of the problem, the proof above assumes that atomic statements $\langle CS \rangle$ and $\langle NCS \rangle$ are always enabled. Once we drop this assumption for statement $\langle NCS_Y \rangle$, the inference from $pc_Y = 1$ to $pc_Y = 2$ is blocked, meaning that something more is needed for progress.

PROOF (of progress at $X.7$ again)
As $pre(Y.1) \Rightarrow \neg y$, we focus attention on showing

$$pc_X = 7 \wedge v \neq X \wedge \neg y \rightsquigarrow pc_X = 8$$

Given the possibility that $Y.1$ may be forever blocked, our only option is to weaken the guard at $X.7$ to $v = X \vee \neg y$ as shown in Figure 9. Weakening the guard in this manner however does not preserve the annotation and leaves $pc_Y \notin C$ a queried assertion at $X.8$. We defer considering how to establish correctness of $pc_Y \notin C$ at $X.8$ until a later stage.

Pre: $\neg x \wedge \neg y \wedge (v = X \vee v = Y) \wedge pc_X = pc_Y = 0$
Invariants: I, J_1, J_2

Component X	Component Y
0: **do** *true* $\rightarrow$	0: **do** *true* $\rightarrow \{\neg y\}$
1: $\langle NCS_X \rangle$;	1: $\langle NCS_Y \rangle$;
2: x:= *true*;	2: y:= *true*;
3: **if** $\neg y \rightarrow \{pc_Y \notin C\}\{x\}$	3: **if** $\neg x \rightarrow$
4: *skip*	4: *skip*
$[]\ y \rightarrow$	$[]\ x \rightarrow$
7: **if** $v = X \vee \neg y \rightarrow$	7: **if** $v = Y \vee \neg x \rightarrow$
$\{?\ pc_Y \notin C\}$	8: *skip*
8: *skip*	**fi**
fi	**fi**;
fi;	5: $\langle CS_Y \rangle$;
5: $\langle CS_X \rangle$;	6: y:= *false* $\parallel$ v:= X
6: x:= *false* $\parallel$ v:= Y	**od**
od	

Figure 9: Refinement 2 (ii)

So again we have a proof of progress, but now the proof relies on the assumption that the guard evaluation statement at $X.7$ is an atomic statement, which, indeed, it is in the programming model described in Section 2.1. Unfortunately, this is not the model that was assumed by Dekker. In Dekker's model, an atomic statement is restricted to at most one access to at most one shared variable. The program in Figure 9 clearly violates this requirement at statements $X.7$ and $X.6$, and the removal of the 'non-atomic' guard evaluation statement $X.7$ is the subject of the next refinement (Figure 10).

Refinement 3

Once again, recalling that set C represents a component about to execute its $\langle CS \rangle$ statement, this refinement induces a further redefinition of C to

$$C \mathrel{\widehat{=}} \{4, 5, 8, 10\}.$$

Again, we appear to have come full circle in this step, returning to the synchronisation statement **if** $\neg y \rightarrow$ *skip* **fi**, but again there is a different context as we have added v to the program state. Having reintroduced the danger of total deadlock at $X.9$ and $Y.9$, we design to avoid this by strengthening the program annotation with P at $X.9$ and $\neg P$ in $Y.9$. The

Pre: $\neg x \wedge \neg y \wedge (v = X \vee v = Y) \wedge pc_X = pc_Y = 0$
Invariants: I, J_1, J_2

Component X	Component Y
0: **do** *true* $\rightarrow$	0: **do** *true* $\rightarrow$
1: $\langle NCS_X \rangle$;	1: $\langle NCS_Y \rangle$;
2: x:= *true*;	2: y:= *true*;
3: **if** $\neg y \rightarrow \{pc_Y \notin C\}\{x\}$	3: **if** $\neg x \rightarrow$
4: *skip*	4: *skip*
$[]\ y \rightarrow$	$[]\ x \rightarrow$
7: **if** $v = X \rightarrow \{v = X\}$	7: **if** $v = Y \rightarrow$
$\{?\ pc_Y \notin C\}$	8: *skip*
8: *skip*	$[]\ v \neq Y \rightarrow \{?\ \neg P\}$
$[]\ v \neq X \rightarrow \{?\ P\}$	9: **if** $\neg x \rightarrow$
9: **if** $\neg y \rightarrow$	10: *skip*
$\{pc_Y \notin C\}\{x\}$	**fi**
10: *skip*	**fi**
fi	**fi**;
fi	5: $\langle CS_Y \rangle$;
fi;	6: y:= *false* $\parallel$ v:= X
5: $\langle CS_X \rangle$;	**od**
6: x:= *false* $\parallel$ v:= Y	
od	

Figure 10: Refinement 3 (i)

new context suggests a choice of P involving v. LC points to $P \equiv v \neq X$, but this is not GC, so we choose $P \equiv (v = X)$ and arrange LC with a second synchronisation statement at $X.11$ as in Figure 11.

Pre: $\neg x \wedge \neg y \wedge (v = X \vee v = Y) \wedge pc_X = pc_Y = 0$
Invariants: I, J_1, J_2

Component X	Component Y
0: **do** *true* $\rightarrow$	$\{\neg y\}$
1: $\langle NCS_X \rangle$;	0: **do** *true* $\rightarrow$
2: x:= *true*;	1: $\langle NCS_Y \rangle$;
3: **if** $\neg y \rightarrow$	2: y:= *true*; $\{y\}$
$\{pc_Y \notin C\}\{x\}$	3: **if** $\neg x \rightarrow$
4: *skip*	4: *skip*
$[]\ y \rightarrow$	$[]\ x \rightarrow \{y\}$
7: **if** $v = X \rightarrow \{v = X\}$	7: **if** $v = Y \rightarrow \{v = Y\}$
$\{?\ pc_Y \notin C\}$	8: *skip*
8: *skip*	$[]\ v \neq Y \rightarrow \{y\}$
$[]\ v \neq X \rightarrow$	11: **if** $v = Y \rightarrow$
11: **if** $v = X \rightarrow$	12: *skip*
12: *skip*	**fi**; $\{v = Y\}$
fi; $\{v = X\}\{x\}$	9: **if** $\neg x \rightarrow$
9: **if** $\neg y \rightarrow$	10: *skip*
$\{pc_Y \notin C\}\{x\}$	**fi**
10: *skip*	**fi**;
fi	5: $\langle CS_Y \rangle$;
fi	6: y:= *false* $\parallel$ v:= X
fi;	**od**
5: $\langle CS_X \rangle$;	
6: x:= *false* $\parallel$ v:= Y	
od	

Figure 11: Refinement 3 (ii)

Refinement 4

Before considering progress at the new synchronisation statement $X.9$, we note that the structure of the code now suggests an opportunity to restore safety at $X.8$. This is done by moving the new synchronisation statement outside of the scope of the conditional statement at $X.7$ as in Figure 12. Note too that the annotation at $X.9$ in Figure 11 still holds at $X.9$ in Figure 12. Again, set C is modified accordingly

$$C \mathrel{\widehat{=}} \{4, 5, 10\}.$$

Pre: $\neg x \wedge \neg y \wedge (v = X \vee v = Y) \wedge pc_X = pc_Y = 0$
Invariants: I, J_1, J_2

Component X	Component Y
0: **do** *true* $\rightarrow$	$\{\neg y\}$
1: $\langle NCS_X \rangle$;	0: **do** *true* $\rightarrow$
2: x:= *true*;	1: $\langle NCS_Y \rangle$;
3: **if** $\neg y \rightarrow$	2: y:= *true*; $\{y\}$
$\{pc_Y \notin C\}\{x\}$	3: **if** $\neg x \rightarrow$
4: *skip*	4: *skip*
$[]\ y \rightarrow$	$[]\ x \rightarrow \{y\}$
7: **if** $v = X \rightarrow$	7: **if** $v = Y \rightarrow$
8: *skip*	8: *skip*
$[]\ v \neq X \rightarrow$	$[]\ v \neq Y \rightarrow \{y\}$
11: **if** $v = X \rightarrow$	11: **if** $v = Y \rightarrow$
12: *skip*	12: *skip*
fi	**fi**
fi; $\{v = X\}\{x\}$	**fi**; $\{v = Y\}$
9: **if** $\neg y \rightarrow$	9: **if** $\neg x \rightarrow$
$\{pc_Y \notin C\}\{x\}$	10: *skip*
10: *skip*	**fi**
fi	**fi**;
fi;	5: $\langle CS_Y \rangle$;
5: $\langle CS_X \rangle$;	6: y:= *false* $\parallel$ v:= X
6: x:= *false* $\parallel$ v:= Y	**od**
od	

Figure 12: Refinement 4

Refinement 5

PROOF (of progress at $X.9$) $pc_X = 9 \rightsquigarrow pc_X = 10$. As before, this goal reduces to

(1) $pc_X = 9 \wedge y \rightsquigarrow pc_X = 9 \wedge \neg y$

(2) $pc_X = 9 \wedge \neg y \rightsquigarrow pc_X = 10$

(2) again confronts us with the problem of a potentially oscillating guard, but we begin with (1), to establish the possibility of Y making the guard true.

$$\begin{array}{ll} & pc_X = 9 \wedge y \rightsquigarrow pc_X = 9 \wedge \neg y \\ \Leftarrow & \\ & \forall\, i :: pc_X = 9 \wedge y \wedge pc_Y = i \rightsquigarrow pc_X = 9 \wedge \neg y \end{array}$$

First, note that $pc_Y = 6$ leads to $\neg y$ by the immediate progress rule. Next, note that the proof is simplified by eliminating control points from consideration. The annotation eliminates $pc_Y \in \{0, 1, 2\}$

$$\begin{array}{ll} & pc_Y \in \{0, 1, 2\} \\ \Rightarrow & \quad \{\text{annotation}\} \\ & \neg y \\ \Rightarrow & \quad \{pc_X = 9 \wedge y\} \\ & \mathit{false} \end{array}$$

and $pc_Y \in \{8, 9, 12\}$ on account of $v = Y$, which leaves $pc_Y \in \{3, 4, 7, 11, 10, 5\}$, of which {4,10,5} lead to 6, which just leaves $pc_Y \in \{3, 7, 11\}$

$$\begin{array}{ll} & pc_Y = 3 \\ \rightsquigarrow & \quad \{\text{Immediate progress as } pc_X = 9 \Rightarrow x\} \\ & pc_Y = 7 \\ \rightsquigarrow & \quad \{\text{Immediate progress as } pc_X = 9 \Rightarrow v = X\} \\ & pc_Y = 11 \end{array}$$

At this point we are stuck at $Y.11$ since the guard is $v = Y$ and $pc_X = 9 \Rightarrow v = X$. A further refinement is needed, and we have only one viable choice for a code change in component Y. Enabling the guard at $Y.11$ with v:= Y at $Y.11$ is clearly not an option. Nor is arranging disjointness of states ($pc_X = 9 \wedge pc_Y = 11 \Rightarrow \mathit{false}$) using x:= *false* at $Y.11$, because it upsets the GC of x in X. This only leaves y:= *false* at $Y.11$, restoring LC of the annotation of Y by adding statement y:= *true* after $Y.11$ as in Figure 13. Carefully recapitulating the proof of (1) will show that it is now concluded.

It remains to prove (2)

Pre: $\neg x \wedge \neg y \wedge (v = X \vee v = Y) \wedge pc_X = pc_Y = 0$
Invariants: I, J_1, J_2

Component X	Component Y
0: **do** *true* →	$\{\neg y\}$
1: $\langle NCS_X \rangle$;	0: **do** *true* →
2: x:= *true*;	1: $\langle NCS_Y \rangle$;
3: **if** $\neg y$ →	2: y:= *true*; $\{y\}$
$\{pc_Y \notin C\}\{x\}$	3: **if** $\neg x$ →
4: *skip*	4: *skip*
[] y →	[] x → $\{y\}$
7: **if** $v = X$ →	7: **if** $v = Y$ →
8: *skip*	8: *skip*
[] $v \neq X$ →	[] $v \neq Y$ →
13: x:= *false*;	13: y:= *false*; $\{\neg y\}$
11: **if** $v = X$ →	11: **if** $v = Y$ →
12: *skip*	12: *skip*
fi;	**fi**;
14: x:= *true*	14: y:= *true* $\{y\}$
fi; $\{v = X\}\{x\}$	**fi**; $\{v = Y\}$
9: **if** $\neg y$ →	9: **if** $\neg x$ →
$\{pc_Y \notin C\}\{x\}$	10: *skip*
10: *skip*	**fi**
fi	**fi**;
fi;	5: $\langle CS_Y \rangle$;
5: $\langle CS_X \rangle$;	6: y:= *false* ‖ v:= X
6: x:= *false* ‖ v:= Y	**od**
od	

Figure 13: Refinement 5 (i)

(2) $pc_X = 9 \wedge \neg y \rightsquigarrow pc_X = 10$.

The annotation of Y shows that only $pc_Y \in \{0, 1, 2, 11\}$ need be considered since these are the only states consistent with $v = X \wedge \neg y$ and, by repeated application of the immediate progress rule, we have

$$\begin{array}{l} \quad pc_X = 9 \wedge pc_Y = 0 \\ \rightsquigarrow \\ \quad pc_X = 10 \vee (pc_X = 9 \wedge pc_Y = 3) \end{array}$$

and, by the proof of (1) above

$$\begin{array}{l} \quad pc_X = 9 \wedge pc_Y = 3 \\ \rightsquigarrow \\ \quad pc_X = 9 \wedge pc_Y = 11 \end{array}$$

and, by immediate progress

$$\begin{array}{l} \quad pc_X = 9 \wedge pc_Y = 11 \\ \rightsquigarrow \\ \quad pc_X = 10 \end{array}$$

It remains to prove that X makes progress at the synchronisation statement at $X.11$.

Proof (of progress at $X.11$) $pc_X = 11 \rightsquigarrow pc_X = 12$

(1) $pc_X = 11 \wedge v \neq X \rightsquigarrow pc_X = 11 \wedge v = X$

(2) $pc_X = 11 \wedge v = X \rightsquigarrow pc_X = 12$

(2) follows by the immediate progress rule as $v = X$ is GC in X. We next check (1) by appealing to the usual rule

$$\begin{array}{l} \quad pc_X = 11 \wedge v \neq X \rightsquigarrow pc_X = 11 \wedge v = X \\ \Leftarrow (\forall\, i :: pc_X = 11 \wedge v \neq X \wedge pc_Y = i \rightsquigarrow \\ \quad pc_X = 11 \wedge v = X) \end{array}$$

As $pc_X = 11$ is stable in Y, by immediate progress we have

$$\begin{array}{l} \quad v \neq X \wedge pc_Y = 0 \\ \rightsquigarrow v \neq X \wedge pc_Y = 1 \\ \quad v \neq X \wedge pc_Y = 2 \\ \rightsquigarrow v \neq X \wedge pc_Y = 3 \qquad \{pc_X = 11 \Rightarrow \neg x\} \\ \rightsquigarrow v \neq X \wedge pc_Y = 4 \\ \rightsquigarrow v \neq X \wedge pc_Y = 5 \\ \rightsquigarrow v \neq X \wedge pc_Y = 6 \\ \rightsquigarrow v = X \end{array}$$

$$\begin{array}{l} \quad v \neq X \wedge pc_Y = 7 \qquad \{v = Y\} \\ \rightsquigarrow v \neq X \wedge pc_Y = 8 \\ \rightsquigarrow v = X \end{array}$$

$$\begin{array}{l} \quad v \neq X \wedge pc_Y = 13 \\ \rightsquigarrow v \neq X \wedge pc_Y = 11 \qquad \{v = Y\} \\ \rightsquigarrow v \neq X \wedge pc_Y = 12 \\ \rightsquigarrow v \neq X \wedge pc_Y = 14 \\ \rightsquigarrow v \neq X \wedge pc_Y = 9 \qquad \{pc_X = 11 \Rightarrow \neg x\} \\ \rightsquigarrow v \neq X \wedge pc_Y = 10 \\ \rightsquigarrow v = X \end{array}$$

Only one transition is missing to complete the proof, the transition from $Y.1$ to $Y.2$, and, as before, since this statement might be continually disabled, we must appeal to the program annotation to permit the inference by disjointness of states.

$$\begin{array}{l} \quad \mathit{false} \\ \equiv \qquad \{v = X \vee y \vee pc_Y \neq 1\} \\ \quad pc_X = 11 \wedge v \neq X \wedge pc_Y = 1 \wedge \neg y \\ \rightsquigarrow \\ \quad pc_X = 11 \wedge v = X \end{array}$$

It remains to check that the program can be correctly annotated with $pre(X.11) = (v = X \vee y \vee pc_Y \neq 1)$. LC is at the cost of the same annotation at $X.13$ and $X.7$. For GC at $X.11$, statements $Y.0$ (falsifies $pc_Y \neq 1$), and $Y.13$ and $Y.6$ (falsify y) need to be considered. GC under $Y.13$ and $Y.6$ is straightforward because $pc_Y \neq 1$ is maintained. For $Y.0$, we have

$$\begin{array}{l} \quad wlp.(pc_Y := 1).(v = X \vee y \vee pc_Y \neq 1) \\ \equiv v = X \vee y \\ \Leftarrow v = X \\ \Leftarrow pc_X = 11 \wedge (v = X \vee pc_X \notin \{7, 13, 11\}) \end{array}$$

The precondition of $Y.0$ is strengthened with $v = X \vee pc_X \notin \{7, 13, 11\}$ accordingly, and we must check that this annotation is correct. LC is by statement $Y.6$ and by *Pre*. For GC at $Y.0$, statements $X.6$ (falsifies $v = X$) and $X.3$ (statement $X.7$ is reached from $X.3$) need to be considered. GC under $X.6$ is straightforward because $pc_X \notin \{7, 13, 11\}$ is maintained. For $X.3$, we have

$$\begin{array}{l} \quad y \Rightarrow wlp.(pc_X := 7).(pc_X \notin \{7, 13, 11\}) \\ \equiv \\ \quad y \Rightarrow \mathit{false} \\ \equiv \qquad \{\text{coassertion } \neg y \text{ at } Y.0\} \\ \quad \mathit{true} \end{array}$$

These changes are reflected in Figure 14.

A final refinement

In Figure 14, the multiple assignment at $X.6$ is non-atomic in the model assumed by Dekker so it is necessary to decompose this multiple assignment into a pair of its component assignments. This is done in Figure 15. The change affects the local correctness of the annotation at $Y.0$. It can be restored by annotating $Y.6$ with $v = X \vee pc_X \notin \{7, 13, 11\}$, but loss of the coassertion $\neg y$ upsets its global correctness under statement $X.3$. A solution is to once more extend the definition of critical set C to represent a

Pre: $\neg x \wedge \neg y \wedge (v = X \vee v = Y) \wedge pc_X = pc_Y = 0$
Invariants: I, J_1, J_2

Component X	Component Y
0: **do** *true* $\rightarrow$	$\{\neg y\}\{v = X \vee pc_X \notin \{7, 13, 11\}\}$
1: $\langle NCS_X \rangle$;	0: **do** *true* $\rightarrow$
2: x:= *true*;	1: $\langle NCS_Y \rangle$;
3: **if** $\neg y \rightarrow$	2: y:= *true*;
$\{pc_Y \notin C\}\{x\}$	3: **if** $\neg x \rightarrow$
4: *skip*	4: *skip*
$[]\ y \rightarrow$	$[]\ x \rightarrow$
$\{v = X \vee y \vee pc_Y \neq 1\}$	7: **if** $v = Y \rightarrow$
7: **if** $v = X \rightarrow$	8: *skip*
8: *skip*	$[]\ v \neq Y \rightarrow$
$[]\ v \neq X \rightarrow$	13: y:= *false*;
13: x:= *false*;	11: **if** $v = Y \rightarrow$
$\{v = X \vee y \vee pc_Y \neq 1\}$	12: *skip*
11: **if** $v = X \rightarrow$	**fi**;
12: *skip*	14: y:= *true*
fi;	**fi**;
14: x:= *true*	9: **if** $\neg x \rightarrow$
fi; $\{v = X\}\{x\}$	10: *skip*
9: **if** $\neg y \rightarrow$	**fi**
$\{pc_Y \notin C\}\{x\}$	**fi**;
10: *skip*	5: $\langle CS_Y \rangle$;
fi	6: y:= *false* $\parallel$ v:= X
fi;	**od**
5: $\langle CS_X \rangle$;	
6: x:= *false* $\parallel$ v:= Y	
od	

Figure 14: Refinement 5 (ii)

component being about to execute its $\langle CS \rangle$ statement with $pc \in \{4, 5, 10\}$ or executing its exit protocol with $pc \in \{15, 6\}$.

$$C \mathrel{\widehat{=}} \{4, 5, 10, 15, 6\}$$

It is routine to check that the annotation is correct, and so satisfies mutual exclusion by construction, and that the proofs of progress at $X.9$ and $X.11$ presented in refinement 4 remain correct. The program derivation is now concluded.

Pre: $\neg x \wedge \neg y \wedge (v = X \vee v = Y) \wedge pc_X = pc_Y = 0$
Invariants: I, J_1, J_2

Component X	Component Y
0: **do** *true* $\rightarrow$	$\{\neg y\}\{v = X \vee pc_X \notin \{7, 13, 11\}\}$
1: $\langle NCS_X \rangle$;	0: **do** *true* $\rightarrow$
2: x:= *true*;	1: $\langle NCS_Y \rangle$;
3: **if** $\neg y \rightarrow$	2: y:= *true*;
$\{pc_Y \notin C\}\{x\}$	3: **if** $\neg x \rightarrow$
4: *skip*	4: *skip*
$[]\ y \rightarrow$	$[]\ x \rightarrow$
$\{v = X \vee y \vee pc_Y \neq 1\}$	7: **if** $v = Y \rightarrow$
7: **if** $v = X \rightarrow$	8: *skip*
8: *skip*	$[]\ v \neq Y \rightarrow$
$[]\ v \neq X \rightarrow$	13: y:= *false*;
13: x:= *false*;	11: **if** $v = Y \rightarrow$
$\{v = X \vee y \vee pc_Y \neq 1\}$	12: *skip*
11: **if** $v = X \rightarrow$	**fi**;
12: *skip*	14: y:= *true*
fi;	**fi**;
14: x:= *true*	9: **if** $\neg x \rightarrow$
fi; $\{v = X\}\{x\}$	10: *skip*
9: **if** $\neg y \rightarrow$	**fi**
$\{pc_Y \notin C\}\{x\}$	**fi**;
10: *skip*	5: $\langle CS_Y \rangle$;
fi	15: v:= X;
fi;	$\{v = X\}\{pc_Y \in C\}$
5: $\langle CS_X \rangle$; $\{pc_Y \notin C\}$ $\{x\}$	6: y:= *false*
15: v:= Y;	**od**
6: x:= *false*	
od	

Figure 15: Final refinement

Overview

Dekker's algorithm has been successfully derived from the safe sluice algorithm in a series of six small refinement steps. This section provides an overview of each of these steps. The first refinement removes deadlock at the synchronisation point $X.3$ in Figure 4. This, however, introduces the danger of violating safety and motivates a new synchronisation statement at $X.7$ in Figure 8. The second refinement is entirely driven by the need to satisfy progress at this newly created synchronisation point. However, progress can only be proved under the assumption that non-critical sections terminate, and the removal of this assumption has the effect of complicating the new synchronisation statement $X.7$ in Figure 9. This complication runs contrary to the model of atomic statements that was assumed by Dekker, and the purpose of the third refinement is to restore this model. Unfortunately, this change reintroduces the danger of total deadlock (at $X.9$ in Figure 10) that arises in the safe sluice algorithm. Fortunately, the new context allows us to remove this danger at the cost of a second synchronisation statement (at $X.11$ in Figure 11). Refinement four is concerned with restoring the mutual exclusion property that was upset at the first refinement step and at this point (Figure 12) we finally achieve a correct program annotation. The fifth refinement is concerned with proving progress at the two synchronisation points ($X.9$ and $X.11$ in Figure 12) that were introduced at refinement three. The sixth, and final, refinement is concerned with decoupling the multiple assignment at $X.6$ in Figure 14. It turns out that there is only one way to do this, and the result is Dekker's algorithm.

5 Conclusion

We have shown how the extended theory of Owicki and Gries can be used to design a concurrent program in a way that gives proper consideration to progress as well as safety requirements. Dekker's program was an ideal choice for this purpose because, as we have seen, so much of its code is bound up with satisfaction of the progress requirement that it is difficult to imagine a derivation being made in the core theory of Owicki and Gries. The example illustrates several tactical aspects of progress driven derivation, an example of which is the common proof pattern that is used to discharge a progress obligation at a synchronisation point, which is illustrated in refinements 1, 2 and 5.

As to the workload of the proof, whilst it is long, this must be traded against the size of each refinement step. Experience shows that it is easy to make mistakes when reasoning about a concurrent program, which recommends a process of small steps in which the burden of checking the correctness of each step is kept managable. Second, the complexity of the program should not be overlooked when assessing the complexity of its proof. Since Dekker's program is inherently complicated, we should not expect to be able to show it correct in a simple fashion, no matter what approach is taken. The reader is once again invited to convince themself by operational means that Dekker's algorithm is correct. Third, we note that derivation leaves open the possibility of alternative design in a way that verification does not, where alternative, possibly better, programs must be overlooked. In fact, going back to our first refinement of Figure 4, we could have produced Peterson's algorithm for two process mutual exclusion instead of Dekker's had we chosen to weaken the guard at the synchronisation statement $X.3$. It is quite satisfying that a derivational approach to programming is able

to isolate the essential difference between these two programs in this way, when they are separated by some 15 years in their publication. Finally, we note that our derivation compares favourably to the formal treatment of a verification of Dekker's algorithm in (Francez 1986).

As to alternative programming models, there are several event based models, such as (Chandy & Misra 1988, Lamport 1994, Lynch & Tuttle 1989, Back & Sere 1989), which really only differ amongst themselves in terms of the ease with which a given program can be formalized in a given model. We see the strength of our approach over these in its ability to support a more direct translation of a program design into code on account of the concurrent sequential programming model that we have adopted. Orthogonal to the choice of programming model is the choice of how to use it. For instance, (Lamport 1994) describes refinement techniques that preserve both safety and progress properties, but where the focus is on the validation of refinements rather than the synthesis of code. (Chandy & Misra 1988) have shown how to derive a UNITY program in a way that takes account of both safety and progress requirements, but, as just remarked, this event-based model is somewhat removed from ours. (Stølen 1990) presents a derivation of Dekker's algorithm in a compositional setting, also addressing progress, however, although the specification is clearly that of Dekker's algorithm, it is unclear how the code itself is generated. (Dingel 1999) describes a refinement method based on program syntax that combines compositional reasoning with the refinement calculus of (Morgan 1990), where both safety and liveness properties are given mention. (Feijen & van Gasteren 1999), as already remarked, derive programs using the same model as ours, but where formal derivation only takes account of safety requirements. Nevertheless, their style of using logic for program design is the approach that we find most appealing, and the one that we have tried to build upon in this paper.

References

Back, R. J. & Sere, K. (1989), Stepwise refinement of action systems, *in* 'International Conference on Mathematics of Program Construction, 375th Anniversary of the Groningen University', pp. 115–138.

Chandy, K. M. & Misra, J. (1988), *Parallel Program Design, A Foundation*, Addison-Wesley.

Dijkstra, E. W. (1976), *A Discipline of Programming*, Prentice Hall.

Dijkstra, E. W. (1982), A personal summary of the Gries-Owicki theory, *in* 'Selected Writings on Computing: A Personal Perspective', Springer-Verlag.

Dingel, J. (1999), A trace-based refinement calculus for shared-variable parallel programs, *in* 'Seventh International Conference on the Algebraic Methodology and Software Technology (AMAST '98) Amazonia, Brazil'.

Dongol, B. & Goldson, D. (2004), Extending the theory of Owicki and Gries with a logic of progress, *in* 'Proceedings of Principles of Software Engineering 2004'.

Feijen, W. H. J. & van Gasteren, A. J. M. (1999), *On a Method of Multiprogramming*, Springer-Verlag.

Francez, N. (1986), *Fairness*, Springer-Verlag.

Lamport, L. (1994), 'The temporal logic of actions', *ACM Transactions on Programming Languages and Systems* **16**(3), 872–923.

Lynch, N. & Tuttle, M. (1989), 'An introduction to input/output automata', *CWI-Quarterly* **2**(3), 219–246.

Misra, J. (2001), *A Discipline of Multiprogramming*, Springer-Verlag.

Morgan, C. (1990), *Programming from Specifications*, Prentice-Hall.

Owicki, S. & Gries, D. (1976), 'Verifying properties of parallel programs: An axiomatic approach', *Communications of the ACM* **19**(5), 279–285.

Stølen, K. (1990), Development of Parallel Programs on Shared Data-structures, PhD thesis, University of Manchester.