Concurrent Program Design in the Extended Theory of Owicki and Gries

Doug Goldson  \hspace{1cm}  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 an 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 verification 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,

\[ S ::= \text{skip} \mid x := E \mid S_0; S_1 \mid \langle S \rangle \mid \text{if } B_0 \rightarrow S_0 \parallel \ldots \parallel B_m \rightarrow S_m \text{ fi} \mid \text{do } B_0 \rightarrow S_0 \parallel \ldots \parallel B_m \rightarrow S_m \text{ od} \]

Here, \( x := E \) \( \equiv x_1 := E_1 \parallel \ldots \parallel x_n := E_n \) 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 \text{if} statement is used to express choice (and condition synchronisation), and a \text{do} statement is used to express repetition.

The atomic statements of the programming language are \text{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 \text{if} or \text{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 (S). 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.

![Figure 1: Safe sluice with labels](image-url)

The essential property of a component labelling 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 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 reason to reason about the correctness of a component. The definition of wlp (Dijkstra 1976) is applied to a labelled statement with implicit program counters pc as follows

2. wlp.(i: x = E || pc:=j) j): P ≡ P[x/E][pc:=j]
3. wlp.(i: (S; pc:=j) j): P ≡ wlp.(S; pc:=j).P
4. wlp.(i: S0; j; S1 k); P ≡ wlp.(i: S0 j); (wlp.(j; S1 k); P)
5. wlp.(i: if (B0 → pc:=j) j): S; S0 l; k; S1 fi k; l; P ≡ (B0 ⇒ wlp.(pc:=j).P[pc:=j].P) \∩ (B1 ⇒ wlp.(pc:=k).P[pc:=k].P)
6. { P } i: do (B → pc:=k) j; S od k { Q } ≡ ((P \∧ B ⇒ wlp.(pc:=j).P[pc:=j].P) \∧ (P \∧ \neg B ⇒ wlp.(pc:=k).P[pc:=k].P))

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 as-

![Figure 2: Safe sluice with program counters](image-url)
also define \( \text{pre}(i) \equiv P \) and \( \text{post}(j) \equiv Q \). 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 \( \text{Pre} \), then \( \text{Pre} \Rightarrow P \)
2. if \( P \) is textually preceded by \( \{ Q \} \) \( S \) \( \{ P \} \) holds.

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

\[
C \equiv \{ 4, 5 \}
\]

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

\[
I \equiv \text{pc}_X \in C \Rightarrow \text{pc}_Y \notin C.
\]

**Proof of safety.** \( I \) is invariant if the annotation in Figure 3 is correct.

![Figure 3: Annotated safe sluice](image)

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

\[
J_1 \equiv \neg y \Rightarrow \text{pc}_Y \notin C
J_2 \equiv \neg x \Rightarrow \text{pc}_X \notin C.
\]

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 \( \text{pc}_Y \in C \) 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 \( \text{pre}(X.4) \) and \( \text{pre}(Y.4) \) from Figure 3. We will need to establish GC of \( \text{pre}(X.4) \), under statement \( Y.3 \) — the if guard evaluation statement in \( Y \), and under statement \( Y.4 \), as they are both able to falsify \( \text{pc}_Y \notin C \) by an implicit update of the program counter in \( Y \). Against \( Y.3, \text{pc}_Y \notin C \) 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 \( \text{pc}_X \notin C \) and we get the correct annotation in Figure 4 on account of

![Figure 4: Safe sluice correctly annotated](image)

**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 \( \Rightarrow \)) relation where \( P \Rightarrow Q \) 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 \( (\text{un}) \). We therefore first present the definition of unless.

**Definition** If \( P \) and \( Q \) are any two predicates, \( P \text{ un} Q \) is true if the Hoare-triple

\[
\{ P \land \neg Q \land A \} \ S \ \{ P \lor Q \}
\]

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 \( \text{un} \) \( Q \) holds for all \( Q \), including false. The rules that define leads-to are now as follows.

![Table: Leads-To Rules](image)
Rule (Immediate Progress Rule) : $P \vdash Q$ holds whenever there is a labelled statement with initial label $i$ in a component with program counter $pc$ and

1. $P \cup Q$

2. $P \land \lnot Q \Rightarrow pc = i$

3. (a) The statement is an assignment or skip statement

$$i : S j : \text{and,} \quad P \land \lnot Q \Rightarrow wp.(i : S j :).Q$$

(b) The statement is an IF statement

$$i : \text{if } B_0 \land j : S_i \text{ and,}$$

$$(i) \quad P \land \lnot Q \Rightarrow B_0 \lor B_i$$

$$(i) \quad (P \land \lnot Q \land B_0 \Rightarrow wp.(pc = j).Q) \land (P \land \lnot Q \land B_i \Rightarrow wp.(pc = k).Q)$$

(c) The statement is a DO statement

$$i : \text{do } B \to j : S \text{ od } k : \text{ and,}$$

$$(P \land \lnot Q \land B \Rightarrow wp.(pc = j).Q) \land (P \land \lnot Q \land \lnot B \Rightarrow wp.(pc = k).Q)$$

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

$$i : (S) j : \text{ and,} \quad P \land \lnot Q \Rightarrow wp.S.(Q[pc = j])$$

Rule (Inductive Progress Rules)

(Transitivity) $P \Rightarrow R \Rightarrow P \Rightarrow Q \land Q \Rightarrow R$

(Disjunction) For any set $W$, $\forall i : i \in W : P_i \Rightarrow Q \Leftrightarrow (\exists i : i \in W : P_i \Rightarrow Q)$

To make sense of these rules we provide these interpretative notes. First of all, we remark that this logic is 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, Dekker’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.”

<table>
<thead>
<tr>
<th>Pre: $\neg x \land \neg y \land (v = X \lor v = Y)$</th>
<th>Component X</th>
<th>Component Y</th>
</tr>
</thead>
<tbody>
<tr>
<td>$0: \text{ do true} \to$</td>
<td>$0: \text{ do true} \to$</td>
<td></td>
</tr>
<tr>
<td>$1: \langle NCS_X \rangle$;</td>
<td>$1: \langle NCS_Y \rangle$;</td>
<td></td>
</tr>
<tr>
<td>$2: x := \text{ true};$</td>
<td>$2: y := \text{ true};$</td>
<td></td>
</tr>
<tr>
<td>$3: \text{ if } \neg y \to$</td>
<td>$3: \text{ if } \neg x \to$</td>
<td></td>
</tr>
<tr>
<td>$4: \text{ skip}$</td>
<td>$4: \text{ skip}$</td>
<td></td>
</tr>
<tr>
<td>$5: \text{ if } v = X \to$</td>
<td>$5: \text{ if } v = Y \to$</td>
<td></td>
</tr>
<tr>
<td>$6: \text{ skip}$</td>
<td>$6: \text{ skip}$</td>
<td></td>
</tr>
<tr>
<td>$7: \text{ if } v \neq X \to$</td>
<td>$7: \text{ if } v \neq Y \to$</td>
<td></td>
</tr>
<tr>
<td>$8: \text{ skip}$</td>
<td>$8: \text{ if } v = X \to$</td>
<td></td>
</tr>
<tr>
<td>$9: \text{ skip}$</td>
<td>$9: \text{ skip}$</td>
<td></td>
</tr>
<tr>
<td>$10: x := \text{ true}$</td>
<td>$10: y := \text{ true}$</td>
<td></td>
</tr>
<tr>
<td>$11: \text{ if } \neg y \to$</td>
<td>$11: \text{ if } \neg x \to$</td>
<td></td>
</tr>
<tr>
<td>$12: \text{ skip}$</td>
<td>$12: \text{ skip}$</td>
<td></td>
</tr>
<tr>
<td>$13: \langle CS_X \rangle$;</td>
<td>$13: \langle CS_Y \rangle$;</td>
<td></td>
</tr>
<tr>
<td>$14: w := Y$;</td>
<td>$14: w := X$;</td>
<td></td>
</tr>
<tr>
<td>$15: x := \text{ false}$</td>
<td>$15: y := \text{ false}$</td>
<td></td>
</tr>
</tbody>
</table>

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 themselves 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” (pp.90-91).

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 ¬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) \( p_C X = 3 \rightarrow p_C X = 4 \) 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. \( p_C X = 3 \land y \rightarrow p_C X = 3 \land \neg y \)
2. \( p_C X = 3 \land \neg y \rightarrow p_C X = 4 \)

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 synchronisation 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 \( (CS) \) statement, the first refinement induces a redefinition of \( C \) to

\[
C \equiv \{4, 5, 8\}.
\]

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

\[
\begin{align*}
K_1 &\equiv B.X \Rightarrow p_C Y \notin C \\
K_2 &\equiv B.Y \Rightarrow p_C X \notin C.
\end{align*}
\]

**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. \( p_C X = 7 \land \neg B.X \rightarrow p_C X = 7 \land B.X \)
2. \( p_C X = 7 \land B.X \rightarrow p_C X = 8 \).

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 \( K_1 \)

\[
B.X \Rightarrow p_C Y \notin C \\
\equiv p_C Y \in C \Rightarrow \neg B.X
\]

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 \( \neg B.X \land X \). 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 synchronisation 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 \( B.X \in GC \) in \( X \). A solution to this equation is to introduce a fresh variable \( v \in \{X, Y\} \) such that

\[
B.X \equiv v = X \\
B.Y \equiv v = Y.
\]

Refinement 2

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

**Proof** (of progress at \( X.7 \)) \( p_C X = 7 \rightarrow p_C X = 8 \) involves showing that

1. \( p_C X = 7 \land v \neq X \rightarrow p_C X = 7 \land v = X \)

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

\[
C \equiv \{4, 5, 8\}.
\]
Fig. 7: Refinement 1 (ii)

(2) $p_{CX} = 7 \land v = X \rightarrow p_{CX} = 8$.

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.

Fig. 8: Refinement 2 (i)

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

$(\forall i: p_{CX} = 7 \land v \neq X \land p_{CY} = i \rightarrow p_{CX} = 7 \land v = X)$.

Noting that $p_{CX} = 7$ is stable in $Y$, by immediate progress we get

$v \neq X \land p_{CY} = 0$
$\neg v \neq X \land p_{CY} = 1$
$\neg v \neq X \land p_{CY} = 2$
$\neg v \neq X \land p_{CY} = 3 \{ p_{CX} = 7 \Rightarrow x \}$
$\neg v \neq X \land p_{CY} = 7 \{ v \neq X \equiv v = Y \}$
$\neg v \neq X \land p_{CY} = 8$
$\neg v \neq X \land p_{CY} = 5$
$\neg v \neq X \land p_{CY} = 6$
$v = X$

$v \neq X \land p_{CY} = 4$
$\neg v \neq X \land p_{CY} = 5$

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 $(CS)$ and $(NCS)$ are always enabled. Once we drop this assumption for statement $(NCS_Y)$, the inference from $p_{CY} = 1$ to $p_{CY} = 2$ 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

$p_{CX} = 7 \land v \neq X \land \neg y \rightarrow p_{CX} = 8$

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

Fig. 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 $(CS)$ statement, this refinement induces a further definition of $C$

$C \equiv \{ 4, 5, 8, 10 \}$.

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

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

\[ C \cong \{4, 5, 10\}. \]

Figure 10: Refinement 3 (i)

Figure 12: Refinement 4

Refinement 5

Proof (of progress at X.9) \( p_{CX} = 9 \leadsto p_{CY} = 10 \). As before, this goal reduces to

1. \( p_{CX} = 9 \land y \leadsto p_{CX} = 9 \land \neg y \)
2. \( p_{CX} = 9 \land \neg y \leadsto p_{CX} = 9 \land \neg y \)

(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.

\[ p_{CX} = 9 \land y \leadsto p_{CX} = 9 \land \neg y \]

First, note that \( p_{CY} = 6 \) 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 \( p_{CY} \in \{0, 1, 2\} \)

\[ \Rightarrow y \quad \text{[annotation]} \]

\[ \Rightarrow \{ p_{CX} = 9 \land y \} \]

\[ \text{false} \]

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

\[ p_{CY} = 3 \]
\[ \leadsto \{ \text{Immediate progress as } p_{CX} = 9 \Rightarrow x \} \]
\[ p_{CY} = 7 \]
\[ \Rightarrow \{ \text{Immediate progress as } p_{CX} = 9 \Rightarrow v = X \} \]
\[ p_{CY} = 11 \]

At this point we are stuck at Y.11 since the guard \( v = Y \) and \( p_{CX} = 9 \Rightarrow v = X \). 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 \( p_{CX} = 9 \land p_{CY} = 11 \Rightarrow \text{false} \) using \( x := \text{false} \) at Y.11, because it upsets the GC of x in X. This only leaves \( y := \text{false} \) at Y.11, restoring LC of the annotation of Y by adding statement \( y := \text{true} \) 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).
\begin{verbatim}
(2) \(pc_X = 9 \land \neg y \implies pc_X = 10\).

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 \(v = X \land \neg y\) and, by repeated application of the immediate progress rule, we have

\[
\begin{align*}
pc_X &= 9 \land pc_Y = 0 \\
\therefore pc_X &= 10 \lor (pc_X = 9 \land pc_Y = 3)
\end{align*}
\]

and, by the proof of (1) above

\[
\begin{align*}
pc_X &= 9 \land pc_Y = 3 \\
\therefore pc_X &= 9 \land pc_Y = 11
\end{align*}
\]

and, by immediate progress

\[
\begin{align*}
pc_X &= 9 \land pc_Y = 11 \\
\therefore pc_X &= 10
\end{align*}
\]

It remains to prove that \(X\) makes progress at the synchronisation statement at \(X.11\).

\textbf{Proof (of progress at \(X.11\))} \(pc_X = 11 \implies pc_X = 12\)

\(1) pc_X = 11 \land v \neq X \implies pc_X = 11 \land v = X\)

\(2) pc_X = 11 \land v = X \implies pc_X = 12\)

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

\[
\begin{align*}
&pc_X = 11 \land v \neq X \implies pc_X = 11 \land v = X \\
&\iff (\forall i: pc_X = 11 \land v \neq X \land pc_Y = i \implies pc_X = 11 \land v = X)
\end{align*}
\]

As \(pc_X = 11\) is stable in \(Y\), by immediate progress we have

\[
\begin{align*}
v &\neq X \land pc_Y = 0 \\
\therefore v &\neq X \land pc_Y = 1
\end{align*}
\]

\vspace{1cm}

Figure 13: Refinement 5 (i)

\vspace{1cm}

\textit{v} \neq X \land pc_Y = 2

\(\therefore v \neq X \land pc_Y = 3\) \{\(pc_X = 11 \implies \neg x\)\}

\(\therefore v \neq X \land pc_Y = 4\)

\(\therefore v \neq X \land pc_Y = 5\)

\(\therefore v \neq X \land pc_Y = 6\)

\(\therefore v = X\)

\(\therefore v \neq X \land pc_Y = 7\) \{\(v = Y\)\}

\(\therefore v \neq X \land pc_Y = 8\)

\(\therefore v = X\)

\(\therefore v \neq X \land pc_Y = 13\)

\(\therefore v \neq X \land pc_Y = 11\) \{\(v = Y\)\}

\(\therefore v \neq X \land pc_Y = 12\)

\(\therefore v \neq X \land pc_Y = 14\)

\(\therefore v \neq X \land pc_Y = 9\) \{\(pc_X = 11 \implies \neg x\)\}

\(\therefore v \neq X \land pc_Y = 10\)

\(\therefore v = X\)

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.

\[
\begin{align*}
false \equiv & \{v = X \lor y \lor pc_Y \neq 1\} \\
& pc_X = 11 \land v \neq X \land pc_Y = 1 \land \neg y \\
& pc_X = 11 \land v = X
\end{align*}
\]

It remains to check that the program can be correctly annotated with \(\text{pre}(X.11) = (v = X \lor y \lor pc_Y \neq 1). \text{LC}\) 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 \(X.13\) and \(X.6\) is straightforward because \(pc_Y \neq 1\) is maintained. For \(Y.0\), we have

\[
\begin{align*}
& \text{wlp.}(pc_Y : 1),(v = X \lor y \lor pc_Y \neq 1) \\
& \equiv v = X \lor y \\
& \equiv v = X \\
& \equiv pc_X = 11 \land (v = X \lor pc_X \neq \{7,13,11\})
\end{align*}
\]

The precondition of \(Y.0\) is strengthened with \(v = X \lor pc_X \neq \{7,13,11\}\) accordingly, and we must check that this annotation is correct. \(LC\) is by statement \(Y.6\) and by \(\text{pre}\). 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

\[
\begin{align*}
& y \Rightarrow \text{wlp.}(pc_X : 7),(pc_X \notin \{7,13,11\}) \\
& \equiv y \Rightarrow false \\
& \equiv \{\text{coassertion } \neg y \text{ at } Y.0\}
\end{align*}
\]

true

These changes are reflected in Figure 14.

\textbf{A final refinement}

In Figure 14, the multiple assignment at \(X.6\) is nonatomic 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 \(v = X \lor pc_X \notin \{7,13,11\}\), 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
component being about to execute its \( \{ \text{CS} \} \) statement with \( PC \in \{ 4, 5, 10 \} \) or executing its exit protocol with \( PC \in \{ 15, 6 \} \).

\[ C \equiv \{ 4, 5, 10, 15, 6 \} \]

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.

---

**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 five 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 \((X.9 \text{ and } X.11 \text{ in Figure 12})\) 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 manageable. 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 himself 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


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


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