Science of Computer Programming 2 (1982) 1–18
North-Holland Publishing Company

A METHODOLOGY FOR PROGRAMMING WITH CONCURRENCY: AN INFORMAL PRESENTATION*

Christian LENGAUER** and Eric C.R. HEHNER
Computer Systems Research Group, University of Toronto, Toronto, Ontario M5S 1A1, Canada

Communicated by M. Sintzoff
Received May 1982

Abstract. In this methodology, programming problems which can be specified by an input/output assertion pair are solved in two steps:

(1) Refinement of a correct program that can be implemented sequentially.

(2) Declaration of program properties, so-called semantic relations, that allow relaxations in the sequencing of the refinement's operations (e.g., concurrency).

Formal properties of refinements comprise semantics (input/output characteristics) and (sequential) execution time. Declarations of semantic relations preserve the semantics but may improve the execution time of a refinement. The consequences are:

(a) The concurrency in a program is deduced from its formal semantics. Semantic correctness is not based on concurrency but precedes it.

(b) Concurrency is a property not of programs but of executions. Programs do not contain concurrent commands, only suggestions (declarations) of concurrency.

(c) The declaration of too much concurrency is impossible. Programs do not contain primitives for synchronization or mutual exclusion.

(d) Proofs of parallel correctness are stepwise without auxiliary variables.

(e) Freedom from deadlock and starvation is implicit without recourse to an authority outside the program, e.g., a fair scheduler.

1. Concepts of concurrent programs

The last decade has brought considerable advances in the field of programming methodology, in general, and in the understanding of concurrency, in particular. The popular technique for programming concurrency is to define a set of concurrent units, *processes*, and to control their interaction by some means of *synchronization*. The early language constructs proposed in the sixties, **fork** and **join** for processes [4] and semaphores for synchronization [5], were intuitive but difficult to formalize. However, the invention of formal methods for the specification of program semantics [13] increased our understanding and ability to handle process programs.

* This research was partially supported by the Social Sciences and Humanities Research Council of Canada and the World University Service of Canada.

** Current address: Department of Computer Sciences, College of Natural Sciences, The University of Texas at Austin, Austin, TX 78712, U.S.A.

Normally, the first step of the program development is to specify, in a *concurrent command*, a set of processes, $S1, S2, \ldots, Sn$,

cobegin $S1||S2|| \ldots ||Sn$ **coend**

which are sequential within themselves but are executed in concurrence with respect to each other. The double slashes ($||$) signify concurrent execution. To prevent concurrency where it may lead to incorrect program behaviours, some construct for *conditional delay* must be added. For instance,

await$\langle B \rightarrow SL \rangle$

appearing in process Si suspends the execution of Si until logical condition B is satisfied. Typically, B will be generated by some concurrent process Sj, $j \neq i$. Upon validation of B, the execution of Si proceeds with statement list SL. While SL is being executed, any concurrent operation that might cause correctness problems (for example, invalidate B again) is suspended. The technique of forcing activities of different concurrent processes into a sequence in order to preserve correctness is called *mutual exclusion*. To protect statement S by mutual exclusion, it is framed with angle brackets:

$\langle S \rangle$.

These constructs for programming with processes (the concurrent command, conditional delay, and mutual exclusion) can be formally defined [21, 28], and we can convince ourselves of the correctness of a process program by

(a) first verifying all processes separately as if they were isolated sequential programs, and then

(b) proving the correctness of their interactions in concurrent execution.

A good example is the proof of a concurrent garbage collector [9].

There are, of course, problems and the one we are particularly concerned with in this paper is the apparent lack of guidelines or criteria to aid the program design. A proof system alone does not necessarily provide support for the development of correct programs. We might continually produce code and discover that it is incorrect. What we need is a *methodology*, a programming calculus that merges program design and verification in order to obtain correct and maybe even particularly suitable programs.

The approach to concurrency just described does not serve as a methodology for program development for the following reasons:

There are no guidelines for the choice of processes. But even if there were, dividing a program into processes is a bold first step, because it usually defines too much concurrency, which then has to be properly pruned with conditional delays and mutual exclusion. Failure to undo all incorrect concurrency leaves one with an incorrect program. Unfortunately, parallel correctness can only be established **after** the development of the processes involved has been completed. Thus the development of processes and the proof of their correct cooperation are strictly

separated. One has to understand all process interactions in their entirety in order to arrive at a correct program.

Our goal is a programming methodology that includes aspects of concurrency. In addition to a formally defined language in which one can communicate programs to the computer, we aim at methods for a correct and suitable stepwise development of such programs. In this methodology, concurrency will be a property not of a program but of an execution. If the semantic properties of the program permit concurrency, an implementation should be able to make use of it to whatever extent is possible and practical. But the semantics will determine the concurrency, not vice versa. Then the correctness of the program does not depend on our understanding of concurrency. A consequence of this view is that our programming language will not contain primitives for sequencing, concurrency, or synchronization. These are aspects of executions, not of the program itself.

We will develop and prove concurrency in steps. Each step will increase the concurrency of the program's executions by observing local properties of some program components. A global understanding of the concurrency permitted by the program is not necessary.

We will be result-oriented, i.e., interested only in results of programs and the speed with which these results can be obtained, but not in certain program behaviours. Our programs will suggest suitable computations rather than expressing a set of given computations. In contrast, process programs are behaviour-oriented, i.e., designed with specific computations in mind.

A much-dreaded sign of the complications parallelism introduces into programs is that the complexity of proofs explodes with increasing concurrency. This is blamed on the necessity to argue consistency of shared data. To prevent such an argument, one can discipline the use of shared variables [14, 29], but that restricts also the potential of concurrency. Or one can eliminate shared variables altogether [8, 16], but may in proofs still have to deal with shared auxiliary variables [1, 24]. *Auxiliary variables* are variables added to a program in order to obtain a proof [28, 29]. Our methodology uses shared variables, and subtle concurrency may require a complex proof. But proofs do not contain auxiliary variables.

There are also the obstacles of deadlock and starvation. Deadlock, the situation where the concurrent execution cannot proceed, can occur in terminating as well as non-terminating applications. Criteria for the prevention or avoidance of deadlock have been investigated [17, 21, 28, 29]. For terminating programs, total correctness implies absence of deadlock. Starvation, the situation where some action concurrent with others can in theory be activated but actually never is, can only occur in non-terminating programs. To avoid starvation one often appeals to a fair scheduler. We shall show that, under certain very simple restrictions, the generalization from terminating to non-terminating programs does not have to pose additional concurrency problems. All programs derived with our methodology will be implicitly free from deadlock and starvation without recourse to an outside authority like a scheduler.

2. Devising a methodology

Our foremost interest is in the result of a program's execution, and in the constraints under which this result can be achieved. In this paper, we do not consider additional constraints on the program's behaviour. Consequently, we want to solve programming problems that can be formally specified by an input/output assertion pair. Because we are looking for results, we do not permit programs to generate the false output assertion. This restricts the range of specifiability to finite problems, i.e., problems that have terminating solutions (the false output assertion indicates non-termination). An infinite problem must be expressed by a specifiable finite segment whose terminating solution can be applied repeatedly. Solutions cannot contain event-driven activities but might be part of a system with real-time constraints. In such a case we may, in addition, specify execution time requirements.

The program development consists of two phases:

(1) Formal refinement of a totally correct program that can be implemented sequentially.

We will use methods that can, if used correctly, only produce refinements whose semantics satisfy the problem specification. The proof of a refinement will also yield a first estimate of its execution time, namely a measure for its sequential execution.

(2) Declaration of program properties, so-called semantic relations, that allow relaxations in the sequencing of the refinement's operations (e.g., concurrency).

We will define semantic relations between refinement components and provide rules for their declaration. Semantic declarations will preserve the semantics of the refinement but may suggest faster executions, e.g., by permitting concurrency.

Semantic declarations are a mechanism for the stepwise development of concurrency. They require only a local understanding of the refinement components appearing in the declared relation. Remember that we are interested in outputs, not in program behaviours. Accordingly, we view concurrency as a tool for satisfying execution time requirements, not for obtaining certain program behaviours.

In the remainder of the paper we give an informal description of the methodology and illustrate it with programming examples. The examples are given without proofs but have been proved (the proofs are contained in the first author's Ph.D. thesis [26]). We are focussing on the overall picture rather than on formal details. A follow-up paper will present the formalism [27] (or the reader may consult the thesis [26]). Unfortunately, people concerned with concurrency are often more interested in the behaviour of programs rather than in a result. They essentially want to simulate activities of concurrency, synchronization, and scheduling. Our methodology does not apply to simulation – we do not specify behaviour (a previous paper contains an attempt to do so [25]). But we can express the desired concurrency, as we informally understand it, in our programming notation.

3. Refinement

The first step of the program development is the derivation of a refinement from the input/output assertion pair that specifies the problem. Our notation for this purpose is RL, a *R*efinement *L*anguage closely related to that of Hehner [11]. Its central feature is a primitive procedure concept, which can be implemented efficiently enough to be used extensively as a refinement mechanism.

Statements in RL can either be refined, or basic (not refined). A refined statement is an invented name, say *S*. Its meaning is conveyed by a refinement, *S*: *SL*, relating the name *S* to a refinement body *SL*. Refinements may be 'indexed', e.g., *Sj*: *SL*, where index *j* is a variable referenced in *SL*. An index is a primitive form of value parameter: *j* may not be changed by *SL*.

In practice, indices will have to be identified by an index declaration in the refinement, e.g., *Sj*(*j*: **int**): *SL*, where *SL* refers to *j*. But, for the sake of brevity, RL does not contain a mechanism for data declarations. We will identify indices as parts of the refinement name that appear in its body.

There are four options of refinement; we call them refinement rules: continuance, replacement, divide (and conquer), and case analysis. Each refinement rule employs a different programming feature: null, assignment, statement composition, or alternation. Divide (and conquer) and case analysis employ a fifth programming feature: the refinement call. Null and assignment are the basic (not refined) statements of RL.

(a) *continuance*:

S: **skip** (null)

skip does nothing at all.

(b) *replacement*:

S: $x := E$ (assignment)

$x := E$ gives variable x the value of expression E.

(c) *divide-in-2*:

S: *S1* ; *S2* (composition)

S1 ; *S2* applies statement *S2* to the results of statement *S1*.

A divide-in-n comprises $n-1$ divide-in-2 in one refinement step. A special case of divide-in-n is the **for** loop. Our notation is

$$\overset{n}{\underset{i=1}{;}}\ Si$$

which stands for *S1* ; . . . ; *Sn*.[1]

[1] The loop bounds must be constants in the loop scope. i is a constant for every step *Si* and local to the loop.

(d) *case analysis*:

S: **if** *B* **then** *S1* **else** *S2* **fi** (alternation)

S1 is applied if *B* evaluates to **true**, *S2* is applied if *B* evaluates to **false**. *B* is called a guard.

We write **if** *B* **then** *S* **fi** for **if** *B* **then** *S* **else skip fi.**

Note the absence of a popular language feature: indefinite repetition. Iterative algorithms are formulated as recursive refinements or, in simple cases, by **for**-composition. While our preference of recursion over repetition is not essential for the methodology, the concept of refinement is.

4. Semantic relations

The second step of the program development is the declaration of semantic relations between parts of the refinement. Semantic relations may allow relaxations in the sequencing of the refinement's operations. The purpose of their declaration is to speed up the refinement's execution, e.g., by concurrency.

It is important to realize that, while this step may improve the execution time of a solution, it is not going to change or add to its semantics. Semantic declarations only make certain semantic properties, which are already laid down in the refinement, more apparent.

We introduce one unary and four binary relations for refinement components. The unary relation is idempotence; the binary relations are commutativity, full commutativity, non-interference, and independence. Refinement components are either statements or guards. Components in general are denoted with the letter *C*, statements in particular with *S*, and guards with *B*. This paper gives only an informal characterization of semantic relations. The precise definitions can be found in [26, 27].

(a) *idempotence*:

!B always,

!S iff *S*; *S* has the same effect as *S*.

An idempotent component *C* may be applied consecutively any number of times.

(b) *commutativity*:

B1 & B2 always,

S & B iff *S* leaves *B* invariant,

S1 & S2 iff *S1*; *S2* has the same effect as *S2*; *S1*.

Commutative components *C1* and *C2* may be applied in any order: *C2* following *C1*, or vice versa.

(c) *full commutativity*:

$C1 \approx C2$ iff $c1 \,\&\, c2$ for all basic components $c1$ in $C1$ and $c2$ in $C2$.

Full commutativity is commutativity rippled down the refinement structure. The execution of fully commutative components $C1$ and $C2$ may be interleaved. Only their basic operations must be indivisible.

(d) *non-interference*:

$C1 \nleftrightarrow C2$ iff every basic component of $C1$ or $C2$ gets no more than one view or update of the set of data shared by $C1$ and $C2$.

Non-interference of components $C1$ and $C2$ lifts the mutual exclusions, i.e., permits the divisibility of their basic components (assignment and guard evaluation).

(e) *independence*

$C1 \| C2$ iff $C1 \approx C2 \wedge C1 \nleftrightarrow C2$.

Independence combines full commutativity with non-interference. Independent components $C1$ and $C2$ may be executed in parallel.

Examples.

(1) $S: x := 3, !S.$

(2) $S: x := y, B: x = 3, \sim(S \,\&\, B), S \nleftrightarrow B.$

(3) $S1: y := x + 1, S2: x := 3, \sim(S1 \,\&\, S2), S1 \nleftrightarrow S2.$

(4) $S: t := p; p := q; q := t, B: p \vee q, S \,\&\, B, \sim(S \approx B), \sim(S \nleftrightarrow B).$

(5) $S1: t := i - 1; i := t; t := i + 1; i := t, S2: j := i, S1 \,\&\, S2, \sim(S1 \approx S2), S1 \nleftrightarrow S2.$

(6) $S1: x := x + a, S2: x := x + b, S1 \approx S2, \sim(S1 \nleftrightarrow S2).$

(7) $S: c := F, B: a \wedge b, S \| B.$

(8) $S1: u := f(w), S2: v := g(w), S1 \| S2.$

Most independence relations will be evident from the following

Independence Theorem. *Two components C1 and C2 of which neither changes any variables appearing in both can be declared independent.*

For a proof see [26, 27].

All independence relations declared in programming examples of this paper are applications of the independence theorem.

5. Semantic declarations

A semantic relation is declared by stating the relation after a refinement. We will only declare idempotence, commutativity, and independence. Relations between guards hold always and do not have to be declared. We will not investigate the question of which relations might be automatically declarable, although it is a very interesting one. For the purpose of this paper, it suffices to assume that the programmer declares every semantic relation (other than between guards).

A set (or complex) declaration, e.g.

$$\{S1, S2\} \| \{T1, T2\}$$

comprises declarations between all set members, in this case,

$$S1 \| T1, \quad S1 \| T2, \quad S2 \| T1, \quad S2 \| T2.$$

A predicate qualifying the range of refinement index values may be used to define the sets involved. For example,

$$\bigwedge_{i,j} pred(i, j): \; Si \| Sj$$

stands for the set of declarations $Si \| Sj$ such that i and j satisfy $pred(i, j)$. A set index in a complex declaration is passed on to all set members. Assume, for instance, the independence of turn signals of different cars in a traffic system: if we give the operations of the left and right turn signals of car i the names $left_i$ and $right_i$, we will declare

$$\bigwedge_{i,j} i \neq j: \{left, right\}_i \| \{left, right\}_j.$$

Semantic declarations define additional computations for the refinement they augment. Computations may contain components (statements and guards), $C1 \rightarrow C2$ indicating sequential execution of $C1$ and $C2$, and $\langle {C1 \atop C2} \rangle$ indicating their parallel execution. We will at this point say no more about the structure of computations and only give a vague idea of the effect of semantic declarations (for details see [26, 27]):

A refinement is characterized by a set of sequential computations. The semantic declarations extend this set as follows:

(a) $!\,C$ adds computations with instances $C \rightarrow C$ replaced by C, and vice versa,

(b) $C1 \,\&\, C2$ adds computations with instances $C1 \rightarrow C2$ or $C2 \rightarrow C1$ swapped,

(c) $C1 \| C2$ adds computations with instances $C1 \rightarrow C2$ or $C2 \rightarrow C1$ replaced by $\langle {C1 \atop C2} \rangle$.

Where further computations can be obtained, the same declarations extend the computation set thus derived, etc. (transitive closure).

Note that semantic relations, although valid, may not be exploitable (i.e., may not generate new computations) for the refinement they are declared for.[2] The declaration and exploitation of semantic relations are separate concerns. We advise to first declare all semantic relations that can be proved, and only in a later step worry about their exploitability.

6. First example: Sorting

The problem is to sort an array $a[0\,..\,n]$ of real numbers into ascending order in time $O(n)$:

sort n. **pre**: $\bigwedge_{i=0}^{n} a[i] \in \mathbf{real}$

sort n. **post**: $\bigwedge_{i,j} (0 \leqslant i < j \leqslant n \supset a'[i] \leqslant a'[j]) \wedge \mathrm{perm}(a, a')$

sort n. **time**: $O(n)$

where perm(a, a') is the predicate that is true if the resulting array a' is a permutation of its original value a, and false otherwise.

Our refinement is an insertion sort adapted from Knuth [20]:

sort n: $\underset{i=1}{\overset{n}{;}} S\,i$

$S0$: **skip**

$(i>0)$ $S\,i$: $cs\,i; S\,i-1$

$cs\,i$: **if** $a[i-1] > a[i]$ **then** *swap i* **fi**

swap i: $t[i] := a[i-1]; a[i-1] := a[i]; a[i] := t[i]$

$\bigwedge_{i,j} j \neq i-1, i, i+1$: $cs\,i \| cs\,j$.

Note that for $|i-j|>1$, $cs\,i$ and $cs\,j$ are disjoint: they do not share any variables. Hence they fulfil the premise of the independence theorem and can be declared independent.

To declare semantic relations for some refinement, one does not need to understand the refinement as a whole. A local understanding of the components appearing in the declared relation is sufficient. Most declarations come easily to mind and have a simple proof.

[2] Example: A conventional **for** loop implementation with incremental step calculation will render semantic relations between loop steps unexploitable. To exploit them, an index value has to be assigned to every step before any step is executed.

For a five-element array ($n=4$), the refinement has the following sequential execution, if we interpret composition ';' as sequencing '→':

$$cs\ 1 \to cs\ 2 \to cs\ 1 \to cs\ 3 \to cs\ 2 \to cs\ 1 \to cs\ 4 \to cs\ 3 \to cs\ 2 \to cs\ 1.$$

If we count the number of comparator modules *cs*, this sequence has length 10, or $n(n+1)/2$ for general n, i.e., it is $O(n^2)$. We can exploit the independence declaration by commuting comparator modules in this sequence left (independence subsumes commutativity), and then 'ravelling' adjacent modules whose indices differ by 2 into parallel:

$$cs\ 1 \to cs\ 2 \to \left\langle \begin{matrix} cs\ 3 \\ cs\ 1 \end{matrix} \right\rangle \to \left\langle \begin{matrix} cs\ 2 \\ cs\ 4 \end{matrix} \right\rangle \to \left\langle \begin{matrix} cs\ 1 \\ cs\ 3 \end{matrix} \right\rangle \to cs\ 2 \to cs\ 1.$$

This execution is of length 7, or $2n-1$ for general n, i.e., $O(n)$. The degree of concurrency increases as we add inputs. We are not limited to a fixed number of concurrent actions. However, if only a fixed number k of processors is available, the independence declaration may not be exploited to generate a degree of concurrency higher than k.

7. Second example: The sieve of Eratosthenes

The odd prime numbers less than a given integer, N, are to be determined by rectifying the initial assumption that all odd numbers are prime. We specify:

sieve. **pre**: $\bigwedge_{i \in I} prime[i]$

sieve. **post**: $\bigwedge_{i \in I} (prime[i] \equiv i \text{ is prime})$

where $I =_{df} \{i \mid 3 \leq i < N, i \in \mathbf{int}, i \text{ odd}\}$.

This is a second 'pure' problem, i.e., a problem which is fully specified by an input/output assertion pair. Our program follows suggestions of Knuth [19]:

sieve: $\mathop{;}_{i=1}^{\lfloor(\sqrt{N}-1)/2\rfloor}$ **if** $prime[2i+1]$ **then** *elim mults of* $2i+1$ **fi**

elim mults of i: $\mathop{;}_{j_i=0}^{\lfloor(N-i^2)/2i\rfloor}$ $prime[i^2+2ij_i] := \mathbf{false}$

(1) $\bigwedge_{i,j} i \neq j$: $prime[i] := \mathbf{false} \| prime[j]$

(2) $\bigwedge_{i,j} i \neq j$: $prime[i] := \mathbf{false} \| prime[j] := \mathbf{false}$

(3) $\bigwedge_{i}$ $!\, prime[i] := \mathbf{false}$.

Any pair of tests and/or eliminations of different numbers is independent; any test or elimination of a number is idempotent. (Remember that guard independence and idempotence need not be declared.)

According to the semantic declarations, guard *prime*[*j*] and statement *elim mults of i* are independent unless *j* happens to be a multiple of *i*. Thus the order of execution of *prime*[*j*] and *elim mults of i* may only be manipulated if *j* is not a multiple of *i*: transformed computations must maintain an order, introduced by the refinement, in which guard *prime*[*j*] appears only where its truth ensures that *j* is prime. In other words, no computation will call *elim mults of i* for non-prime *i*. Moreover, because of idempotence, there are computations which eliminate every multiple exactly once.

The execution time of refinement *sieve* is exponential in the input N. The fastest computations generated by the semantic declarations have an execution time constant in N: the time it takes to eliminate one multiple.

One purpose of this example is to demonstrate how difficult optimum concurrency can be – even when derived from simple semantic declarations. To derive the fastest computations for every input n, all multiples would have to be known – which makes the execution of *sieve* obsolete. Moreover, there is no way to find all multiples other than by infinite enumeration.

But we can compile *sieve* with limited concurrency. We know, for instance, that, if the multiples of the first m primes have been eliminated, the $m+1$st and $m+2$nd element left in I must be prime: the $m+1$st certainly is, the $m+2$nd must be because between a prime and its first multiple there is always another prime.[3] Therefore operations *elim mults of i* can, with increasing *i*, at least proceed pairwise in parallel.[4] We leave it up to the number theorists to find more compilable concurrency.

8. Third example: The dining philosophers

Five philosophers, sitting at a round table, alternate between eating and thinking. When a philosopher gets hungry, he picks up two forks next to his plate and starts eating. There are, however, only five forks on the table, one between each two philosophers. So a philosopher can eat only when neither of his neighbours is eating. When a philosopher has finished eating, he puts down his forks and goes back to thinking. This example demonstrates how the methodology can be used to build never-ending algorithms.

Our methodology can only yield terminating refinements. Therefore we modify the problem specification and give the philosophers a finite life of N meals (allowing different philosophers individually many eating sessions is a trivial extension). If

[3] Bertrand's Postulate [10].

[4] For a process program exploiting this fact see Hoare [15].

variable $eaten[i]$ counts the meals of philosopher i and variable $fork[i]$ indicates if fork i is currently on the table ($fork[i]=0$) or not ($fork[i]=1$), the semantic specification becomes:

$$lives.\ \textbf{pre}: \quad \bigwedge_{i=0}^{4} fork[i]=0$$

$$lives.\ \textbf{post}: \quad \bigwedge_{i=0}^{4} (fork'[i]=0 \wedge eaten'[i]=eaten[i]+N).$$

Resulting values of *fork* and *eaten* are primed, initial values are unprimed.

We represent the philosophers' actions by statements up_i and $down_i$ for a movement, i.e., seizure and release of fork i, eat_i for a meal, and $think_i$ for a thinking period of philosopher i:

$$lives: \quad {}^{N}_{1};\!\left[\!\left[{}^{4}_{i=0};\!\right] phil_i \right]\!\!\right]$$

$phil_i$: $up_i; up_{i\oplus 1}; eat_i; down_i; down_{i\oplus 1}; think_i$

up_i: $fork[i]:=1$

$down_i$: $fork[i]:=0$

eat_i: $use\ fork\ i; use\ fork\ i\oplus 1; eaten[i]:=eaten[i]+1$

$use\ fork\ i$: **if** $fork[i]=1$ **then skip else** *error* **fi**

$think_i$: **skip**

(1) $\bigwedge_{i,j} j\neq i$: $\quad phil_i \,\&\, phil_j$

(2) $\bigwedge_{i,j} j\neq i\ominus 1, i, i\oplus 1$: $\quad eat_i \| eat_j$

(3) $\bigwedge_{i,j} j\neq i, i\oplus 1$: $\quad eat_i \| \{up, down\}_j$

(4) $\bigwedge_{i,j} j\neq i$: $\quad \{up, down\}_i \| \{up, down\}_j$

(5) $\bigwedge_{i,j} j\neq i$: $\quad think_i \| phil_j.$

$\oplus$ denotes addition, $\ominus$ subtraction modulo 5; $use\ fork\ i$ is an operation which touches variable $fork[i]$ but has no effect. For *error* any refinement can be chosen – it should never be executed. Thinking philosophers do nothing of relevance to the system.

This program lets philosophers properly compete for their share of the meal and eventually die. The declarations state that philosophers may eat at different intervals according to their hunger (1), non-neighbours may eat at the same time (2), forks

that are presently not used for eating may be moved (3), different forks may be moved in parallel (4), and thinking philosophers do not interact with the rest of the system (5). There are additional, undesirable semantic relations in this refinement whose declaration we carefully avoided. For instance, a philosopher can think at any time, not just between eating sessions. For a more complicated refinement of $phil_i$ which does not contain undesirable semantic relations see [26].

The total correctness of the refinement guarantees that the system cannot get stuck. None of the five semantic declarations invalidates total correctness (no correct declaration ever does), and therefore the concurrent version is also deadlock-free. We do not need additional proof, but to help convince the reader, here is a reasoning especially tailored for this algorithm. In the refinement, before concurrency is considered, philosopher 0, say, completes his eating session, putting down his forks, before philosopher 1 picks up his forks. The declarations do not permit philosopher 0's final $down_1$ to commute with philosopher 1's initial up_1. Entire eating sessions ($phil_i$) can be commuted, and when non-neighbours are adjacent, their eating sessions can be concurrent. But no declaration can generate a deadlocked execution (where each philosopher has one fork and is waiting for the other) from an execution without deadlock.

For a never-ending program, a solution to the infinite problem, the finite *lives* may be called repeatedly in sequence. The user of our finite algorithms is responsible for a mechanism for infinite repetition; we refuse to consider it in our calculus. But we guarantee that, if the problem specification allows an infinite repetition of the solution (i.e., if the output assertion implies the input assertion), all algorithmic properties except termination are preserved. The user can rely on partial correctness, absence of deadlock, and absence of starvation without recourse to an authority outside the program, e.g., a fair scheduler.

Our solutions may be slightly more restrictive than non-terminating solutions have to be. We do not allow unbounded non-determinism, not even for unbounded activities. In this example, the table is cleared completely in arbitrarily long intervals, whereas it is not necessary for all philosophers to leave (or, in our terminology, die).

There is a straight-forward extension to our calculus which lifts this restriction: the exploitation of semantic relations between components of **different** calls of refinement *lives* must be allowed. Then a call of *lives* can start before the previous call terminates. Deadlock remains impossible, but to prevent starvation, a fair scheduler must ensure that no call $phil_i$ is commuted to infinity.

9. Fourth example: Producing and consuming

Suppose that two parts of a program are almost distinct; there is only one variable, *buf*, that appears in both. One part 'produces' values and deposits them into the buffer, the other 'consumes' deposited values by reading them from the buffer. The following program fragment reflects this situation. Productions and consumptions

are called in turn M times; the omitted expressions in the assignments represent the creation of item i in production i, and the use of item i in consumption i:

$$\begin{aligned} &stream: && \mathop{;}_{i=0}^{M-1} [prod_i; cons_i] \\ &prod_i: && buf := \cdots \\ &cons_i: && \cdots := buf. \end{aligned}$$

The dependence of productions and consumptions forces an execution in sequence. But we would like some concurrency. We apply the independence theorem (Sect. 4) and disconnect different *prod*/*cons* pairs by giving everyone its private buffer. Then a production of some item i may go on in parallel with the consumption of other items j produced previously:

$$\begin{aligned} &stream: && \mathop{;}_{i=0}^{M-1} [prod_i; cons_i] \\ &prod_i: && buf[i] := \cdots \\ &cons_i: && \cdots := buf[i] \\ (1)\quad &\bigwedge_{i,j} i \neq j: && prod_i \| cons_j. \end{aligned}$$

We did not specify any special properties of productions and consumptions other than their interaction over *buf*. If the creation of item i does not depend on knowledge of items 0 to $i-1$, productions may be mutually independent and so may consumptions. Then we may add two more declarations:

$$(2)\quad \bigwedge_{i,j} i \neq j: \quad prod_i \| prod_j$$

$$(3)\quad \bigwedge_{i,j} i \neq j: \quad cons_i \| cons_j.$$

For an implementation we must assume a bound, say n, on the number of variables. We modify the previous program by indexing the buffer modulo n:

$$\begin{aligned} &stream: && \mathop{;}_{i=0}^{M-1} [prod_i; cons_i] \\ &prod_i: && buf[i \textbf{ mod } n] := \cdots \\ &cons_i: && \cdots := buf[i \textbf{ mod } n] \\ (1)\quad &\bigwedge_{i,j} (i \neq j) \textbf{ mod } n: && prod_i \| cons_j \end{aligned}$$

and again, if productions are mutually independent and so are consumptions,

$$(2)\quad \bigwedge_{i,j} (i \neq j) \textbf{ mod } n: \quad prod_i \| prod_j$$

(3) $\bigwedge_{i,j} (i \neq j) \bmod n: \quad cons_i \| cons_j.$

Concurrency works only for *prod/cons* pairs within a certain neighbourhood:[5] productions can be at most n items ahead of consumptions, then the buffer is full; consumptions can at most catch up with productions, then the buffer is empty. Note that the condition 'buffer empty' is part of any program with producing and consuming, whereas 'buffer full' arises out of a need for a buffer bound in an implementation.

The outcome of this section is nothing new. The solution to concurrent producing and consuming is well known. But we wanted to exhibit its stepwise development in our methodology and tried not to rely on previous knowledge.

10. Summary

We take the view that concurrency is a property not of programs but of executions. Consequently, looking at programs, concurrency may be hard to recognize. RL programs do not contain concurrent commands or synchronization primitives, only declarations of independence. And because an independence declaration may remain unexploited it only suggests that some action may or may not be involved in a parallel execution.

The reader might feel that starting with the definition of processes helps structuring a program, and that a refinement without immediate regard to concurrency forces us to artificially order logically separate tasks. We believe there is a separation of concerns: modularity structures the problem solution, concurrency speeds its execution up. For example, an airline reservation or taxi dispatching system receives its structure from the modularity of its transactions. It can well be imagined as an arbitrary sequence of transactions (in fact, this view will be helpful), although only a concurrent execution will be practical.

Our methodology yields modularity by way of refinement (as part of the language!) and concurrency by way of semantic declarations. A change in the refinement is likely to affect the semantic declarations for it (but an extension does not). Concurrency works bottom-up and is therefore susceptible to top-down design changes. But we insist the solutions are modifiable. They only put concurrency in its proper place: determined by, not determining the semantics of the refinement. If a refinement does not permit enough parallelism, the independence theorem advises us to spread out variables over its components.

Concurrent actions are not synchronized by conditional delays but by conditional concurrency. The solutions are the same but our methodology prevents overdefinition and subsequent restriction of concurrency. The definition of concurrency proceeds step by step on semantically correct territory, successive declarations

[5] More independence is declared but not exploitable.

yielding faster and faster executions. Exclusion is not explicitly programmed. A process design approaches a solution from incorrect territory by trying to exclude wrong concurrency.

In our methodology proofs do not require auxiliary variables [28, 29]. It seems auxiliary variables have the purpose of relating the histories of concurrent program parts that have been proved separately, like processes. In our programs, concurrency is not synthesized from separate histories. (This is not obvious from the present paper; see [26, 27].)

We consider only programming problems with results, i.e., we can only derive terminating programs. Infinite repetition is a mechanism of the user environment and applies only to entire RL programs, not their parts. Absence of deadlock is guaranteed by the total correctness of the RL program, and the question of starvation does not arise.

We do not permit program properties to vary due to overlapped execution. Consider the following program of two processes whose outcome depends on the point at which the first process is executed:

cobegin $\langle x := 1 \rangle$ // **do** $\langle x = 0 \rightarrow y := y + 1 \rangle$ **od coend.**

There is no equivalent in RL because different interleaved executions have different properties.[6] A popular programming problem of this type is concurrent garbage collection as described for a LISP environment by Dijkstra [6] and Gries [9], and RL in its present form can consequently not express it. (This is explained better in [26, 27].) An extension to include such programs would seriously complicate the formal theory of the methodology.

11. Related research

The introduction and summary relate our methodology to customary programming with processes. We have referred to the proof methods for process programs by Owicki and Gries [28, 29]. These methods are, however, not concerned with program development.

The view of programming with concurrency closest to ours is taken by van Lamsweerde and Sintzoff [22]. They use a concurrent command but begin with sequential semantics. Exclusions may be removed by way of correctness-preserving program transformations which are not described in detail. The concurrent processes

[6] Note that limiting the number of repetitions,

do k **times** $\langle x = 0 \rightarrow y := y + 1 \rangle$ **od**

or skipping the assignment to y,

do $\langle x = 0 \rightarrow$ **skip**$\rangle$ **od**

still does not yield programs expressible in *RL*: the point of execution of $x := 1$ determines both the partial correctness and termination of the **do** loop.

are guarded commands (transitions). The guards (synchronizing conditions) make any order necessary for correctness explicit. Transitions are repeated forever, and non-trivial formal techniques are necessary to prevent deadlock and starvation.

Broy describes the semantics of concurrent programs by correctness-preserving transformations but elaborates only on the opposite direction: serializing concurrent statements [3]. The transformations exploit a simple global independence relation.

Idempotence appears in a previous paper of the second author [12], but for programs with traditional concurrency features (concurrent commands and synchronization primitives). The implementation described there works only in the presence of concurrency and only for last-action calls.

Path expressions [7], incorporated in the specification language for concurrent systems COSY [23] and recently extended to predicate path expressions [2], reflect an approach inverse to ours: starting out with complete concurrency one declares relations, paths, that tighten sequencing. But since no quality distinctions are made between different sequences, there is no selection problem. As a specification tool path expressions do not come with a proof system.

Jones recognizes the lack of development methods for programs with concurrency, he calls them 'interfering programs', and proposes extensions to previous methodologies for sequential programs, what he calls 'isolated programs' [18]. The basic idea is to incorporate requirements for parallel correctness into the problem specification.

References

[1] K.R. Apt, N. Francez and W.P. de Roever, A proof system for communicating sequential processes, *ACM TOPLAS* **2** (3) (1980) 359–385.

[2] S. Andler, Predicate path expressions, *Proc. 6th Annual Symposium on Principles of Programming Languages* (1979) 226–236.

[3] M. Broy, Transformational semantics for concurrent programs, *Information Processing Lett.* **11** (2) (1980) 87–91.

[4] M.E. Conway, A multiprocessor system design, *AFIPS Conference Proceedings* **24**, FJCC 63 (1963) 139–146.

[5] E.W. Dijkstra, Co-operating sequential processes, in: F. Genuys, Ed., *Programming Languages* (Academic Press, London, 1968) 43–112.

[6] E.W. Dijkstra et al., On-the-fly garbage collection: An exercise in cooperation, *Comm. ACM* **21** (11) (1978) 966–975.

[7] L. Flon and A.N. Habermann, Towards the construction of verifiable software systems, *Proc. 2nd International Conference on Software Engineering* (1976) 141–148.

[8] D.I. Good, R.M. Cohen and J. Keeton-Williams, Principles of proving concurrent programs in Gypsy, *Proc. 6th Annual Symposium on Principles of Programming Languages* (1979) 45–52.

[9] D. Gries, An exercise in proving parallel programs correct, *Comm. ACM* **20** (12) (1977) 921–930; Corrigendum: *Comm. ACM* **21** (12) (1978) 1048.

[10] G.H. Hardy and E.M. Wright, *An Introduction to the Theory of Numbers* (Oxford University Press, London 4th ed., 1980) 343.

[11] E.C.R. Hehner, **do** considered **od**: A contribution to the programming calculus, *Acta Informat.* **11** (4) (1979) 287–304.

[12] E.C.R. Hehner, On the design of concurrent programs, *INFOR* **18** (4) (1980) 289–299.

[13] C.A.R. Hoare, An axiomatic basis for computer programming, *Comm. ACM* **12** (10) (1969) 576–580, 583.
[14] C.A.R. Hoare, Monitors: An operating system structuring concept, *Comm. ACM* **17** (10) (1974) 549–557; Corrigendum: *Comm. ACM* **18** (2) (1975) 95.
[15] C.A.R. Hoare, Parallel programming: An axiomatic approach, *Comput. Languages* **1** (2) (1975) 151–160.
[16] C.A.R. Hoare, Communicating sequential processes, *Comm. ACM* **21** (8) (1978) 666–677.
[17] R.C. Holt, Some deadlock properties of computer systems, *Comput. Surveys* **4** (3) (1972) 179–196.
[18] C.B. Jones, Tentative steps towards a development method for interfering programs, Programming Research Group, Oxford University (1980).
[19] D.E. Knuth, *The Art of Computer Programming, Vol. 2: Seminumerical Algorithms* (Addison-Wesley, Reading, MA, 1969) 360; or: 2nd ed. (1981) 394.
[20] D.E. Knuth, *The Art of Computer Programming, Vol. 3: Searching and Sorting* (Addison-Wesley, Reading, MA, 1973) 220ff.
[21] L. Lamport, Proving the correctness of multiprocess programs, *IEEE Trans. Software Engrg.* **3** (2) (1977) 125–143.
[22] A. van Lamsweerde and M. Sintzoff, Formal derivation of strongly correct concurrent programs, *Acta Informat.* **12** (1) (1979) 1–31.
[23] P.E. Lauer, P.R. Torrigiani and M.W. Shields, COSY—A system specification language based on paths and processes, *Acta Informat.* **12** (2) (1979) 109–158.
[24] G.M. Levin and D. Gries, A proof technique for communicating sequential processes, *Acta Informat.* **15** (3) (1981) 281–302.
[25] C. Lengauer and E.C.R. Hehner, A methodology for programming with concurrency, in: W. Händler, Ed., *CONPAR 81*, Lecture Notes in Computer Science **111** (Springer, Berlin, 1981) 259–270.
[26] C. Lengauer, A methodology for programming with concurrency, Technical Report CSRG-142, Computer Systems Research Group, University of Toronto (1982).
[27] C. Lengauer, A methodology for programming with concurrency: the formalism, *Sci. Comput. Programming* **2** (1) (1982) 19–52.
[28] S. Owicki and D. Gries, An axiomatic proof technique for parallel programs I, *Acta Informat.* **6** (4) (1976) 319–340.
[29] S. Owicki and D. Gries, Verifying properties of parallel programs: An axiomatic approach, *Comm. ACM* **19** (5) (1976) 279–285.