A METHODOLOGY FOR PROGRAMMING WITH CONCURRENCY

Christian Lengauer

Eric C.R. Hehner

Abstract

A programming calculus is presented which will yield programs with simple, suitable, and safe concurrency.

The program design consists of three steps:

(1) specification of a finite problem by a pre/post condition pair
(2) formal refinement of a totally correct solution which can be implemented sequentially
(3) declaration of program properties which allow relaxations in sequencing (→ concurrency)

For infinite problems programs can be executed repeatedly without correctness problems. For the choice of refinement formal guidelines can be used.

The derived solutions have the following properties:

(a) dynamic concurrency - processes do not have to be statically declared
(b) only correct concurrency can be specified - exclusion is not explicitly programmed
(c) conditional concurrency - no conditional delays
(d) stepwise proofs of parallel correctness without auxiliary variables
(e) simply derived freedom from deadlock and starvation without appealing to a fair scheduler

---

Address of authors:
University of Toronto, Computer Systems Research Group, 121 St. Joseph's St.,
Toronto, Ontario, Canada, M5S 1A1

Research supported by the Social Sciences and Humanities Research Council of Canada

1 Introduction

The last decade has brought considerable advances in the field of programming methodology, in general, and in the understanding of concurrency and its problems, in particular.

The popular technique for programming concurrency is to define in a special kind of statement a set of concurrent units, processes, and to control their interaction by some means of synchronization. The first language constructs proposed in the sixties, fork and join for processes [Con63] and semaphores for synchronization [Dij68], were intuitive but difficult to formalize. However, the invention of formal methods for the specification of program semantics [Hoa69] increased our understanding and ability to handle process programs. According to [OwGr76a, Lam77],

(a) we first verify all processes separately as if they were sequential programs,

(b) we then prove the correctness of their interactions on shared data by a so-called non-interference argument.

A good example is the proof of a concurrent garbage collector [Gri77].

There are, of course, problems which have partly been pointed out by the inventors themselves. The most striking one is that the complexity of proofs explodes with increasing concurrency. This is blamed on the necessity to argue non-interference. To prevent interference one can either impose restrictions on concurrency features in the language [OwGr76b] or eliminate process interaction on shared variables altogether [Hoa78, GCW79].

Another problem is the lack of support for a suitable design of algorithms. One has to understand all process interactions in their entirety in order to arrive at a correct solution and can only learn from the experience gained in previous unsuccessful attempts of verification.

Our goal is a programming methodology that can handle concurrency. In addition to a formally defined language in which one can communicate algorithms to the computer, we aim at methods for a correct and suitable development of such algorithms. We want to be problem-oriented, i.e., build algorithmic solutions to programming problems without regard to implementational considerations not specified in the problem. Such solutions will suggest a suitable machine rather than being adapted to an available machine, but will be executable on real machines.

Finally, there are 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 absence or avoidance of deadlock have been investigated [Hol72, OwGr76b, Lam77]. 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 the generalization from terminating to non-terminating algorithms does not have to pose additional concurrency problems. All algorithms derived with our methodology will be implicitly free from deadlock and starvation.

2 Devising a Methodology

We split the development of a program into three phases:

(1) Specification of a finite problem by a pre/post condition pair.

Because we aim at total correctness, we exclude the false postcondition. This restricts the range of specifiability to finite problems, those which have terminating solutions. But infinite problems can also be solved: we can specify pre- and postconditions for finite subproblems and execute their solutions repeatedly.

Solutions cannot contain event-driven activities but might be part of a system with real-time constraints. In such a case we may specify additional execution speed requirements but will not do so in this paper. We may also specify subproblems and add concurrency requirements for their solutions.

(2) Formal refinement of a totally correct solution which can be implemented sequentially.

We will use refinement methods which can, if used correctly, only produce correct results. There will also be some help for a suitable choice of correct refinement.

(3) Declaration of program properties which allow relaxations in sequencing (→ concurrency).

We will define semantic relations between program components and provide rules for their declaration.

In the remainder of the paper we give a formal description of the methodology and an illustration with programming examples. The examples are given without formal proofs, but their correctness is simple and can be understood informally.

3 The Refinement Language

We use the weakest precondition calculus to describe the semantics of programs. The weakest precondition for statement S with respect to postcondition R (written wp(S,R) in [Dij76]) is here denoted $S\{R\}$.

We use the logical symbols T (true), F (false), $\wedge$ (and), $\vee$ (or), $\sim$ (not), $\supset$ (implies), and the quantifiers $\bigwedge_i$ (for all i) and $\bigvee_i$ (there exists i).

R_E^x is R with every free occurrence of variable x replaced by expression E.

$S\{R\}_P^{s\{\}}$ is $S\{R\}$ as derived when P is used in place of the weakest precondition for substatement s.

The central feature of our refinement language is a built-in refinement mechanism [Heh79]. Statements may 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 to a refinement list SL. Refinements may be indexed, e.g., Sj: SL , where j is a variable referenced in SL. An index is a primitive form of value parameter.

To ensure correctness, refinements can only be defined according to four refinement rules:

Consider a problem specification (P,R). To obtain a totally correct refinement named S such that $P \supset S\{R\}$ we may choose one of the following:

(R1) continuance: if $P \supset R$ then choose S: skip

(R2) replacement: if $P \supset R^x_E$ then choose S: x:=E

(R3) divide-2: if $\bigvee_Q \bigvee_{S1,S2} P \supset S1\{Q\},\ Q \supset S2\{R\}$ then choose: S: S1;S2

A divide-n is easily formulated and comprises n-1 divide-2 in one refinement step. A special case of divide-n is the for loop. We have a special notation: for example, $\overset{5}{\underset{i=1,2}{;}} Si$ stands for S1;S3;S5.

(R4) case analysis: if $\bigvee_{B1,..Bn} \bigvee_{S1,..Sn} (P \supset B1 \vee ... \vee Bn \wedge \overset{n}{\underset{i=1}{\wedge}} P \wedge Bi \supset Si\{R\})$
then choose S: if B1 → S1 [] ... [] Bn → Sn fi

The construct Bi → Si is called a guarded command [Dij76]. Bi is guard for alternative Si. We write if B then S fi for if B → S [] ~B → skip fi.

Rules (R3) and (R4) ask for further refinements. For more details on their proper choice see the notion of progress in [Heh79].

Five language rules define the features which may be used in a refinement:

For all predicates R,

(L1) null: $\underline{skip}\{R\} = R$

(L2) assignment: $x:=E\{R\} = R^x_E$

(L3) concatenation: $S1;S2\{R\} = S1\{S2\{R\}\}$

(L4) alternation: $\underline{if}\ B1 \to S1\ [] \ ...\ []\ Bn \to Sn\ \underline{fi}\{R\} = \bigvee_i Bi \wedge \bigwedge_i (Bi \supset Si\{R\})$

(L5) refinement call: for refinement S: SL,

(a) non-recursive: $S\{R\} = SL\{R\}$

(b) (singly) recursive:

SL_{rest} denotes that part of the refinement which succeeds the recursive call (i.e., in a proof for SL the precondition of SL_{rest} is postcondition of S). With

$$Q_0 = R$$

$$k \hat{>} 0 \qquad Q_k = SL_{rest}\{Q_{k-1}\}$$

$$k \hat{\geq} 0 \qquad S_0\{Q_k\} = F$$

$$i \hat{>} 0 \quad k \hat{\geq} 0 \qquad S_i\{Q_k\} = SL\{Q_k\}^{S\{\}}S_{i-1}\{Q_{k+1}\}$$

the axiom states

$$S\{R\} = \overset{\vee}{i \geq 0}\ S_i\{Q_0\}$$

This subsumes the tail recursion rule of [Heh79] (which subsumes Dijkstra's do...od repetition rule [Dij76]): for tail recursion, all Q_k are R.

For indexed refinements we use a simple substitution rule: with formal index list $\bar{y}$ and actual index list $\bar{c}$,

$$S\bar{c}\{R\} = S\bar{y}\{R\}^{\bar{y}}_{\bar{c}}$$

To avoid proving a refinement for different postconditions R, calls can be related to a single refinement proof for, say, postcondition Q (Q should not refer to indices or local variables of S): if $\bar{z}$ is the list of global variables of S and $\bar{u}$ ranges over the values of $\bar{z}$ which establish Q,

$$(\ S\bar{y}\{Q\}^{\bar{y}}_{\bar{c}}\ \wedge\ \underset{\bar{u}}{\wedge}\ (\ Q^{\bar{z}}_{\bar{u}} \supset R^{\bar{z}}_{\bar{u}}\ )\ ) \supset S\bar{c}\{R\}$$

For more details and a still simpler call rule see [GrLe80].

Language rules (L1) to (L4) are taken from [Dij76, Heh79]. Rule (L5) is a formalization of some remarks in [Heh79].

4 The Semantic Relations

We define one unary and three binary relations for program components, specifically, statements or guards. Components in general are denoted with the letter C, statements in particular with S, and guards with B. R ranges over all predicates but may be restricted to just the predicates of a particular proof which are relevant:

(a) idempotence (permits variation of the number of executions)

idem B iff T

idem S iff $S\{R\} = S;S\{R\}$ for all R

(b) commutativity (permits arbitrary order of execution)

B1 com B2 iff T

S com B iff $B \wedge S\{R\} = S\{B \wedge R\}$ for all R

S1 com S2 iff $S1;S2\{R\} = S2;S1\{R\}$ for all R

(c) full commutativity (permits interleaved execution)

C1 fcom C2 iff c1 com c2 for all components c1 of C1 and c2 of C2

(d) independence (permits concurrent execution)

C1 ind C2 iff C1 fcom C2 and C1 ⫽ C2

where we choose as operational requirement for non-interference, under the assumption of memory interlock [Gri77],

C1 ⫽ C2, read: (the executions of) C1 and C2 do not interfere

iff any expression E in C1 contains at most one reference to at most one variable changed in C2;
if C1 contains x:=E and C2 references x, then E does not refer to x nor to any variable changed by C2;
and vice versa.

Examples:

(1) S: collect_garbage, idem S
(2) S: x:=y, B: x=3, ~(S com B)
(3) S1: x:=3, S2: y:=x+1, ~(S1 com S2)
(4) S: swap(p,q), B: p∨q, S com B, ~(S ind B)
(5) S1: x:=x+a, S2: x:=x+b, S1 com S2, ~(S1 ind S2)
(6) S: c:=*F*, B: a∧b, S ind B
(7) S1: u:=f(w), S2: v:=g(w), S1 ind S2

5 The Semantic Declarations

A semantic relation is declared by stating the relation. Relations which hold always, such as between guards, do not have to be declared. (In this category belong also relations involving skip, and relations between S and ~B if already declared between S and B. Therefore the hidden guarded command in an if-then can be neglected for semantic declarations.)

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

{S1,S2} ind {T1,T2}

comprises declarations between all set members, in this case,

S1 ind T1, S1 ind T2, S2 ind T1, S2 ind T2

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

$\hat{\imath}\ \hat{\jmath}$ pred(i,j): Si ind Sj

stands for the set of declarations Si ind Sj such that i and j satisfy pred(i,j). A set index in a complex declaration is passed on to all set members. For instance, the independence of turn signals of different cars in a traffic system may be

declared as

$$\bigwedge_i \bigwedge_j i\neq j:\ \{\text{left,right}\}_i \ \underline{\text{ind}}\ \{\text{left,right}\}_j$$

Semantic declarations define additional implementable computations for the refinement they augment. The effect can here be only informally described:

The proof for a refinement S yields a set of sequential computations. The semantic declarations extend this set as follows: take some computation† for S, then

(a) idem C adds computations with instances C↠C replaced by C, or vice versa,
(b) C1 com C2 adds computations with instances C1↠C2 or C2↠C1 swapped,
(c) C1 fcom C2 adds computations with instances C1↠C2 or C2↠C1 interleaved,
(d) C1 ind C2 adds computations with instances C1↠C2 or C2↠C1 in parallel.

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) in the refinement they are declared for. We advise to declare all concurrency that can be proved whether or not it is exploitable.

6 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: $\bigwedge_{i\in M}$ prime[i]

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

where $M = \{ i \mid 3\leq i<N, i \text{ odd} \}$

This example demonstrates how the concurrency permitted in an algorithm can be declared. We refine a solution proposed by Knuth [KnuII,p.360]:

sieve: $\underset{i=3,2}{\overset{\sqrt{N}}{;}}$ if prime[i] then elim mults of i fi

elim mults of i: $m_i := i^2$; elim mults of i from m_i on

elim mults of i from m_i on: prime[m_i]:=*F* ; $m_i := m_i+2i$;
if $m_i<N$ then elim mults of i from m_i on fi

(1) $\bigwedge_i \bigwedge_j i\neq j$: elim mults of i ind elim mults of j

(2) $\bigwedge_i \bigwedge_j i\neq j$: prime[i]:=*F* ind prime[j]

(3) $\bigwedge_i \bigwedge_j$ $\{m_i := i^2, m_i := m_i+2i\}$ ind prime[j]

(4) $\bigwedge_j$ idem prime[j]:=*F*

† Computations may contain statements and guards with ↠ as sequencing operator.

According to (2) and (3) guard "prime[j]" and statement "elim mults of i" are independent unless j happens to be a multiple of i. Because of this exclusion (see the qualifying predicate of (2)) executions must maintain an order, introduced by the refinement, in which guard "prime[j]" appears only where its truth ensures that j is prime: no execution will call "elim mults of i" for non-prime i.

Note that there is no limit to the number of concurrent activities. (Compare Hoare's attempt of a parallel solution with a concurrent statement defining processes [Hoa75].)

7 Producing and Consuming

An arbitrary finite number, M, of items are to be produced and consumed.

This example demonstrates how concurrency requirements can be made for an operating systems application. We break the problem down into subproblems:

$prod_i$.pre:	T	$cons_i$.pre:	item i produced
$prod_i$.post:	item i produced	$cons_i$.post:	item i produced and consumed

stream.pre: T

stream.post: all M items produced and consumed

stream.con: at any time, at least k prod/cons pairs proceed concurrently (or all pairs left to be executed, if less than k)

The assignments in the solution are left incomplete because we do not want to regard the nature of the items, only their transfer. We presume items are independent.

stream: $\underset{i=0}{\overset{M-1}{;}} [\![prod_i ; cons_i]\!]$

$prod_i$: buf[i]:=

$cons_i$: :=buf[i]

(1) $\underset{i}{\wedge}\underset{j}{\wedge} i\neq j$: $prod_i$ ind $cons_j$

(2) $\underset{i}{\wedge}\underset{j}{\wedge} i\neq j$: $prod_i$ ind $prod_j$

(3) $\underset{i}{\wedge}\underset{j}{\wedge} i\neq j$: $cons_i$ ind $cons_j$

To satisfy a concurrency requirement one has to declare independence and show that it is exploitable. This program satisfies stream.con for every k>0 : all declared independence is exploitable.

To permit concurrency, the refinement decouples different prod/cons pairs by giving them separate variables for communication. We employed the following refinement guideline:

> To render two components independent without appealing to any hardware support, let neither access any variable the other changes.

For an implementation we must assume a bound, say n, on the number of variables. We modify the previous solution by indexing modulo n. This requires a new proof and new semantic declarations:

stream: $\overset{M-1}{\underset{i=0}{;}} [\![\text{prod}_i ; \text{cons}_i]\!]$

prod_i: buf[i|n]:=

cons_i: :=buf[i|n]

(1) $\hat{i}\hat{j}$ (i≠j)|n: prod_i ind cons_j

(2) $\hat{i}\hat{j}$ (i≠j)|n: prod_i ind prod_j

(3) $\hat{i}\hat{j}$ (i≠j)|n: cons_i ind cons_j

As long as n≥k, stream.con remains satisfied.

The development reflects that the synchronization on "buffer empty" is problem-inherent, whereas a synchronization on "buffer full" arises out of a need for a buffer bound in an implementation. The concurrency structure of our solutions is problem-oriented: prod/cons independence is on the same level as prod/prod and cons/cons independence, in contrast with process solutions [OwGr76a].

8 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 only eat 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. For the problem specification we give the philosophers a finite life of N eating sessions:

lives.pre: *T*

lives.post: every philosopher has eaten N times during his life

We model the lives with basic statements "up_i" and "down_i" for a movement, i.e., seizure and release of fork i, and "eat_i" for an eating session of philosopher i[†]:

lives: $\overset{N}{\underset{1}{;}} [\![\overset{4}{\underset{i=0}{;}} \text{phil}_i]\!]$

phil_i: up_i ; $\text{up}_{i\oplus 1}$; eat_i ; down_i ; $\text{down}_{i\oplus 1}$

[†] ⊕ denotes addition modulo 5

(1) $\wedge_i \wedge_j$ j≠i: phil_i com phil_j

(2) $\wedge_i \wedge_j$ j≠i⊖1,i,i⊕1: eat_i ind eat_j

(3) $\wedge_i \wedge_j$ j≠i,i⊕1: eat_i ind $\{\text{up,down}\}_j$

(4) $\wedge_i \wedge_j$ j≠i: $\{\text{up,down}\}_i$ ind $\{\text{up,down}\}_j$

Thinking sessions are not included. Thinking philosophers do not interact with the rest of the system. Consider the lag time between two activations of "phil_i" as thinking†.

This solution lets the philosophers properly compete for their share of the meal and eventually die. The declarations state that philosophers may eat in 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), and different forks may be moved in parallel (4).

The total correctness of the refinement guarantees that the system cannot get stuck. None of the four semantic declarations invalidates total correctness, and therefore the concurrent program is deadlock-free. We do not need additional proof, but to help the reader being convinced, here is a reasoning especially tailored for this algorithm: a situation where every philosopher has one fork and waits for the other cannot arise because in the refinement philosopher i lets no neighbour access the forks next to him once he prepares for eating, and none of the declarations performs commutations which would lift this restriction. The key is that (3) does not commute eating sessions and the movement of forks for neighbours.

For a never-ending program, a solution to the infinite problem, the finite "lives" may be called repeatedly. The user of our finite algorithms is responsible for a mechanism for infinite repetition, and we refuse to add it to our notation . But we guarantee that, if the problem specification allows infinite repetition of the solution, all algorithmic properties except termination are preserved. The user can rely on partial correctness, absence of deadlock, and absence of starvation.

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).

† A solution which models thinking sessions exists but has more complicated semantic declarations.

9 Conclusions

The actions executed concurrently in our programs have not much in common with those in process programs. They are not static: because an independence declaration may remain unexploited in some computation it only suggests that some action may or may not be involved in a concurrent execution. Also, there is no bound on the number of actions concurrent with each other.

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. 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.

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

Finally (although not demonstrated in this paper), in our methodology proofs do not require auxiliary variables in the algorithms [OwGr76a,76b].

10 Further Research

The presented material is only a fragment of the topics considered in the first author's Ph.D. thesis to appear.

Semantic declarations global to a program are too restrictive for many applications. More involved definitions exist which tie semantic relations to specific post- and preconditions. The effects of semantic declarations on the set of computations for a refinement are being formalized, and ways of selecting a fast computation are being investigated.

A more powerful methodology that handles problem specifications with execution speed requirements is also in preparation.

11 References

[Con63] Conway, M.E.
A Multiprocessor System Design
AFIPS FJCC 24 (1963), 139-146

[Dij68] Dijkstra, E.W.
Co-operating Sequential Processes
in "Programming Languages", F. Genuys (Ed.), Academic Press, 1968, 43-112

[Dij76] Dijkstra, E.W.
A Discipline of Programming
Prentice-Hall, Series in Automatic Computation, 1976, 217 p.

[GCW79] Good, D.I.; Cohen, R.M.; Keeton-Williams, J.
Principles of Proving Concurrent Programs in Gypsy
Proc. Principles of Programming Languages 1979, 42-52

[Gri77] Gries, D.
An Exercise in Proving Parallel Programs Correct
Comm. ACM 20, 12 (Dec 77), 921-930
Corrigendum: Comm. ACM 21, 12 (Dec 78), 1048

[GrLe80] Gries, D.; Levin, G.
Assignment and Procedure Call Proof Rules
ACM TOPLAS 2, 4 (Oct 80), 564-579

[Heh79] Hehner, E.C.R.
do considered od: A Contribution to the Programming Calculus
Acta Informatica 11 (1979), 287-304

[Hoa69] Hoare, C.A.R.
An Axiomatic Basis for Computer Programming
Comm. ACM 12, 10 (Oct 69), 576-580, 583

[Hoa75] Hoare, C.A.R.
Parallel Programming: An Axiomatic Approach
Computer Languages 1, 2 (June 75), 151-160

[Hoa78] Hoare, C.A.R.
Communicating Sequential Processes
Comm. ACM 21, 8 (Aug 78), 666-677

[Hol72] Holt, R.C.
Some Deadlock Properties of Computer Systems
ACM Computing Surveys 4, 3 (Sept 72), 179-196

[KnuII] Knuth, D.E.
The Art of Computer Programming, Vol. 2: Seminumerical Algorithms
Addison-Wesley, 1969, 624 p.

[Lam77] Lamport, L.
Proving the Correctness of Multiprocess Programs
IEEE Trans. on Soft. Eng. SE-3, 2 (Mar 77), 125-143

[OwGr76a] Owicki, S.S.; Gries, D.
An Axiomatic Proof Technique for Parallel Programs I
Acta Informatica 6 (1976), 319-340

[OwGr76b] Owicki, S.S.; Gries, D.
Verifying Properties of Parallel Programs: An Axiomatic Approach
Comm. ACM 19, 5 (May 76), 279-285