Concurrency

Eric C.R. Hehner

Department of Computer Science, University of Toronto
hehner@cs.utoronto.ca

Presented in Austin, Texas, on 2016 April 29,
as an invited talk at the celebration of the work of Jayadev Misra on the occasion of his retirement.

Abstract: This paper surveys the ways in which concurrency has been described formally.

Introduction

The treatment of concurrency is divided into two camps, which I will call the conjunction camp and the interleaving camp.

In general terms (details will come later), the conjunction camp says that if A describes an activity that happens starting at time t, and B describes some other activity that happens starting at time t, then their conjunction $A \wedge B$ describes the two activities happening concurrently starting at time t.

The interleaving camp says that to describe activities A and B happening concurrently, you must divide A into its atomic actions A_0, A_1, A_2, ..., and likewise divide B into its atomic actions B_0, B_1, B_2, ..., and then make a sequence of atomic actions by interleaving them; for example, $A_0\, B_0\, A_1\, A_2\, B_1\, B_2 \ldots$.

People in the conjunction camp sometimes call their version "true" concurrency, because it describes activities that really do happen at the same time. By implication, that makes the interleaving camp "false" concurrency, because it really describes a sequence of activities, not concurrent activities. But I don't want to be pejorative. At my slow human speed of perception, the interleaved activities on my computer do appear to be concurrent. And interleaving is the only possible strategy when you have to implement concurrency on a single processor. "True" concurrency requires as many processors as there are processes. But these are just the two extreme points on a range of points. In the middle, you have more than one processor, but not as many as you have processes. So you have to be able to schedule the processes on the processors in a sensible way.

One issue of interleaving is to decide what are the atomic actions, which are the units of activity that get scheduled. Another issue is called "fairness", which means scheduling the atomic actions so that each process gets a fair share of processor time. I have never liked the definition of fairness that uses the words "infinitely often", or "always eventually" because it makes fairness and unfairness unobservable, and therefore imposes no constraint on the scheduler. To me it would make more sense to define fairness in a comparative way, so that one schedule is more fair or less fair than another schedule. Or fairness could be defined in a quantitative way, so that a schedule has a fairness rating. These issues do not arise if concurrency is conjunction.

Concurrency as conjunction applies not only to programs, but also to specifications, which enables programming by refinement, with proof of correctness at each programming step. A specification may not say what the programming activities will be, so interleaving does not apply to specifications. For that reason, it seems to me that concurrency as conjunction is specification-oriented, and interleaving is implementation-oriented.

a Practical Theory of Programming

In my theory of programming [0], we do not specify programs; we specify computation, or computer behavior. The free variables of the specification represent whatever we wish to observe about a computation, such as the initial values of variables, their final values, their intermediate values, interactions during a computation, the time taken by the computation, the space occupied by the computation. Observing a computation provides values for those variables. When you put the observed values into the specification, there are two possible outcomes: either the computation satisfies the specification, or it doesn't. So a specification is a binary expression. If you write anything other than a binary expression as a specification, such as a pair of predicates, or a predicate transformer, you must say what it means for a computation to satisfy a specification, and to do that formally you must write a binary expression anyway.

A program is an implemented specification. It is a specification of computer behavior that you can give to a computer to get the specified behavior. I also refer to any statement in a program, or any sequence or structure of statements, as a program. Since a program is a specification, and a specification is a binary expression, therefore a program is a binary expression. For example, if the program variables are x and y, then the assignment program $x := y+1$ is the binary expression $x'=y+1 \wedge y'=y$ where unprimed variables represent the values of the program variables before execution of the assignment, and primed variables represent the values of the program variables after execution of the assignment.

We can connect specifications using any binary operators, even when one or both of the specifications are programs. If A and B are specifications, then $A \Rightarrow B$ says that any behavior satisfying A also satisfies B, where $\Rightarrow$ is implication. This is exactly the meaning of refinement. As an example, using again integer program variables x and y,

$$x := y+1 \quad \Rightarrow \quad x'>y$$

We can say "$x := y+1$ implies $x'>y$", or "$x := y+1$ refines $x'>y$", or "$x := y+1$ implements $x'>y$". When we are programming, we start with a specification that is not a program, and refine it until we obtain a program, so we may prefer to write

$$x'>y \quad \Leftarrow \quad x := y+1$$

using reverse implication, or "is implied by", or "is refined by", or "is implemented by".

Using the same program variables x and y, and time variable t, the sequential composition $A;B$ of specifications A and B is defined as follows.

$$A;B \;=\; \exists x'', y'', t'' \cdot \quad (\text{for } x', y', t' \text{ substitute } x'', y'', t'' \text{ in } A) \\ \wedge\, (\text{for } x, y, t \text{ substitute } x'', y'', t'' \text{ in } B)$$

Sequential composition of A and B is mainly the conjunction of A and B, but the final state and time of A are identified with the initial state and time of B.

Concurrent composition is defined as follows. Without time, to define $A \| B$, partition the primed variables between A and B, and then

$$A \| B \;=\; A \wedge B$$

With time, partition the variables as before, but the time variable t belongs to both processes. Then

$$A \| B \;=\; \exists t_A, t_B \cdot \quad (\text{for } t' \text{ substitute } t_A \text{ in } A) \\ \wedge\, (\text{for } t' \text{ substitute } t_B \text{ in } B) \\ \wedge\; t' = t_A \uparrow t_B$$

Concurrent composition of A and B is mainly the conjunction of A and B, but the final time is the maximum of the final times of A and B.

Laws

This definition gives us all the laws that we expect for concurrency: it's symmetric, associative, and distributes in the right ways. And monotonicity is essential for programming by refinement.

$P \parallel Q \;=\; Q \parallel P$	symmetry
$P \parallel (Q \parallel R) \;=\; (P \parallel Q) \parallel R$	associativity
$P \parallel Q \vee R \;=\; (P \parallel Q) \vee (P \parallel R)$	distributivity
$P \vee Q \parallel R \;=\; (P \parallel R) \vee (Q \parallel R)$	distributivity
$P \parallel \textbf{if } b \textbf{ then } Q \textbf{ else } R \textbf{ fi} \;=\; \textbf{if } b \textbf{ then } P \parallel Q \textbf{ else } P \parallel R \textbf{ fi}$	distributivity
$\textbf{if } b \textbf{ then } P \parallel Q \textbf{ else } R \parallel S \textbf{ fi} \;=\; \textbf{if } b \textbf{ then } P \textbf{ else } R \textbf{ fi} \parallel \textbf{if } b \textbf{ then } Q \textbf{ else } S \textbf{ fi}$	distributivity
$P \Rightarrow Q \;\Longrightarrow\; (P \parallel R) \Rightarrow (Q \parallel R)$	monotonicity
$P \Rightarrow Q \;\Longrightarrow\; (R \parallel P) \Rightarrow (R \parallel Q)$	monotonicity

A consequence of the definition of sequential composition is the Substitution Law, which is a generalization of the proof rule for assignment in Hoare Logic.

$x := e;\; P \;=\;$ (for x substitute e in P)

It says an assignment followed by any specification is that specification with a substitution. This law is just a special case of a more general law saying that a concurrent composition of assignments followed by any specification is that specification with concurrent substitutions.

$(x := e \parallel y := f);\; P \;=\;$ (for x substitute e and concurrently for y substitute f in P)

Partition

Suppose there are two variables x and y, and consider

$x := x+1 \parallel y := y+1$

Without partitioning,

$x := x+1 \;=\; x' = x+1 \wedge y' = y$

and

$y := y+1 \;=\; y' = y+1 \wedge x' = x$

so their conjunction is *false*, which is unimplementable. With partitioning, giving x' to the left process and y' to the right process,

$x := x+1 \;=\; x' = x+1$

and

$y := y+1 \;=\; y' = y+1$

so their conjunction is $x' = x+1 \wedge y' = y+1$, which is what we want.

Here is another example. Giving x' to the left process and y' to the right process,

$x := y \parallel y := x \;=\; x' = y \wedge y' = x$

We see that x and y swap values. When a process uses the initial value of a variable that's not in its part of the partition, it has to make a private copy.

In this next example, again x' is in the left part of the partition, and y' is in the right part.

$(x := x+y;\; x := x \times y) \parallel (y := x-y;\; y := x/y)$

$= \quad x' = (x+y) \times y \;\wedge\; y' = x/(x-y)$

The left part makes two assignments to x, and in both assignments, y refers to the initial value of variable y. Likewise in the two assignments to y on the right, x refers to the initial value of variable x. But suppose we want the second occurrence of y on the left to refer to the updated value after the first assignment on the right. And likewise we want the second occurrence of x on the right to refer to the updated value after the first assignment on the left. The usual solution is to treat the variables as shared, so each process can see the current value of the variables being changed by the other process. Shared variables require careful timing calculations so that no process tries to read or write a variable while another process is writing

it. Alternatively, variables must be protected by a mutual exclusion mechanism. Also, you need to synchronize the concurrency at the two semi-colons, to make sure that the second assignment in each process waits until the first assignment in the other process is finished. But really, we don't need the headache of shared variables, and we don't need new synchronization primitives. All we need to do is say what we meant.

$(x{:=}\ x{+}y \parallel y{:=}\ x{-}y);\ (x{:=}\ x{\times}y \parallel y{:=}\ x/y)$

$= \quad x' = (x{+}y){\times}(x{-}y) \ \wedge\ y' = (x{+}y)/(x{-}y)$

We meant that the first assignment to x and the first assignment to y are in parallel, and then, sequentially following that, the second assignments to x and y are in parallel. The need for shared memory and synchronization was just a symptom of writing the wrong program.

Programming by Refinement

We can refine a specification as a sequential composition of specifications. For example, again in integer variables x and y,

$y' \geq y{+}2 \ \Leftarrow\ x'{>}y;\ y'{>}x$

(This is not a sensible way to refine $y' \geq y{+}2$; the example is just meant to illustrate refinement by a sequential composition.) The importance of defining sequential composition for all specifications, not just for programs, is that we can prove this refinement before we refine the two new specifications $x'{>}y$ and $y'{>}x$. That way we find an error as soon as it is made, and save the work of further programming on top of an error. Here is the proof.

$x'{>}y;\ y'{>}x$

$= \quad \exists x'', y'', t'' \cdot \quad$ (for x', y', t' substitute x'', y'', t'' in $x'{>}y$)

$\qquad\qquad \wedge$ (for x, y, t substitute x'', y'', t'' in $y'{>}x$)

$= \quad \exists x'', y'', t'' \cdot\ x''{>}y \wedge y'{>}x''$

$= \quad y' \geq y{+}2$

Now we refine each of the two new specifications.

$x'{>}y \ \Leftarrow\ x{:=}\ y{+}1$

$y'{>}x \ \Leftarrow\ y{:=}\ x{+}1$

Thanks to the monotonicity of sequential composition and the transitivity of implication, we have

$y' \geq y{+}2 \ \Leftarrow\ x{:=}\ y{+}1;\ y{:=}\ x{+}1$

for free, with no need to prove it. Some people call this “compositionality”, and others call it “stepwise refinement”.

If we want to be able to use concurrent composition in programming, it too must be monotonic, and must apply to all specifications, not just programs. For example, we might make the refinement

$x'{+}y' > x{+}y \ \Leftarrow\ x'{>}x \parallel y'{>}y$

Now we want to prove this refinement before refining the two new specifications $x'{>}x$ and $y'{>}y$. Here is the proof.

$x'{>}x \parallel y'{>}y$

$= \quad x'{>}x \wedge y'{>}y$

$\Rightarrow \quad x'{+}y' > x{+}y$

Now that we know that step is correct, we can refine $x'{>}x$ and $y'{>}y$.

$x'{>}x \ \Leftarrow\ x{:=}\ x{+}1$

$y'{>}y \ \Leftarrow\ y{:=}\ y{+}1$

Thanks to the monotonicity of concurrent composition and the transitivity of implication, we have

$x'{+}y' > x{+}y \ \Leftarrow\ x{:=}\ x{+}1 \parallel y{:=}\ y{+}1$

without further proof.

We can go a long way with this kind of concurrency. As an example, we'll find the maximum item in a list. Here is the specification, including its execution time.

$$\mathit{findmax}\ i\ j \quad = \quad i<j \ \Rightarrow\ L'i = \Uparrow L[i;..j] \ \wedge\ t' = t+\mathit{log}(j-i)$$

Specification *findmax* , with parameters i and j , says that we find the maximum item of a nonempty segment ($i<j$) of list L from index i to (not including) index j . The time is the logarithm (base 2) of the length of the segment. The specification is refined like this.

$$\begin{aligned}\mathit{findmax}\ i\ j \quad \Leftarrow \quad & \textbf{if}\ j-i>1\ \textbf{then}\ (\mathit{findmax}\ i\ (\mathit{div}\ (i+j)\ 2)\ \|\ \mathit{findmax}\ (\mathit{div}\ (i+j)\ 2)\ j);\\ & \qquad L\ i := L\ i \uparrow L\ (\mathit{div}\ (i+j)\ 2);\ t := t+1\ \textbf{fi}\end{aligned}$$

Find the maximum in the first half of the segment, and concurrently, find the maximum in the second half. Then take the maximum of those two maximums. The time variable is increased by 1 each iteration; it is there for proof purposes, not for execution. This refinement is a reverse implication whose proof is the proof of correctness, and proof of the time bound, and it is also the program.

This simple kind of concurrency is good for a lot of programming, but there are times when you need more interaction than it offers. For that purpose, [0] adds communication channels between processes without changing the definition of concurrency. But I don't pursue that here.

Merge

"Concurrency with merge" is conjunction, but strengthened to make it potentially more useful. Strengthening also makes it harder to implement. There is no need to partition the state space. Instead, you make as many copies of the state space as you have processes, so that each process can work concurrently on its own copy. Then when the last process is finished, the results of all processes are merged together. There are many ways to do the merging; in [2] Hoare and He list ten ways.

In the first edition of [0], merging is done as follows. For each variable, if all processes leave it unchanged, then its final value is unchanged. If just one process changes it, then its final value is that changed value. If more than one process changes it, then its final value is arbitrary. Here is the formal definition, in state variables x and y and time variable t .

$$\begin{aligned}P \| Q \ = \ & \exists x_P, x_Q, y_P, y_Q, t_P, t_Q \cdot\\ & \quad (\text{for } x', y', t' \text{ substitute } x_P, y_P, t_P \text{ in } P)\\ & \wedge\ (\text{for } x', y', t' \text{ substitute } x_Q, y_Q, t_Q \text{ in } Q)\\ & \wedge\ (x_P{=}x \Rightarrow x'{=}x_Q) \wedge (x_Q{=}x \Rightarrow x'{=}x_P)\\ & \wedge\ (y_P{=}y \Rightarrow y'{=}y_Q) \wedge (y_Q{=}y \Rightarrow y'{=}y_P)\\ & \wedge\ t' = t_P \uparrow t_Q\end{aligned}$$

It's still $P \wedge Q$, and the time is still the maximum of the individual process times, and the rest is talking about how to merge the results.

Here is a further strengthening, with the hope of making it even more useful, and the cost of making it even harder to implement. In the previous version, if two or more processes change the value of a variable, then its final value is arbitrary. In this version, if all processes changing the value of a variable agree on its final value, then that is its final value. Here is the formal definition.

$$\begin{array}{lll} P \| Q & = & \exists x_P, x_Q, y_P, y_Q, t_P, t_Q \cdot \\ & & \quad (\text{for } x', y', t' \text{ substitute } x_P, y_P, t_P \text{ in } P) \\ & & \wedge\ (\text{for } x', y', t' \text{ substitute } x_Q, y_Q, t_Q \text{ in } Q) \\ & & \wedge\ (x_P{=}x \Rightarrow x'{=}x_Q) \wedge (x_Q{=}x \Rightarrow x'{=}x_P) \wedge (x_P{=}x_Q \Rightarrow x'{=}x_P) \\ & & \wedge\ (y_P{=}y \Rightarrow y'{=}y_Q) \wedge (y_Q{=}y \Rightarrow y'{=}y_P) \wedge (y_P{=}y_Q \Rightarrow y'{=}y_P) \\ & & \wedge\ t' = t_P \uparrow t_Q \end{array}$$

We can strengthen the definition further, saying that the processes don't have to agree; the final value of a variable is any one of the changed values. Here is the formal definition.

$$\begin{array}{lll} P \| Q & = & \exists x_P, x_Q, y_P, y_Q, t_P, t_Q \cdot \\ & & \quad (\text{for } x', y', t' \text{ substitute } x_P, y_P, t_P \text{ in } P) \\ & & \wedge\ (\text{for } x', y', t' \text{ substitute } x_Q, y_Q, t_Q \text{ in } Q) \\ & & \wedge\ (x_P{=}x \Rightarrow x'{=}x_Q) \wedge (x_Q{=}x \Rightarrow x'{=}x_P) \wedge (x'{=}x_P \vee x'{=}x_Q) \\ & & \wedge\ (y_P{=}y \Rightarrow y'{=}y_Q) \wedge (y_Q{=}y \Rightarrow y'{=}y_P) \wedge (y'{=}y_P \vee y'{=}y_Q) \\ & & \wedge\ t' = t_P \uparrow t_Q \end{array}$$

All of these definitions of concurrency with merge have the attraction that, in principle, you don't have to partition the state space. They form a strengthening sequence, more and more deterministic, saying more and more about the final values of variables. Again, in principle, the more you can say about the final values of variables, the more you potentially benefit computationally from the use of concurrent composition.

On the other side of the ledger, the definitions in this sequence are more and more complicated. More complicated definitions are harder to use in proofs. And the definitions in this sequence are harder and harder to implement, with slower and slower execution.

History and Acknowledgements

I first defined concurrency as conjunction in 1981 [1], and I continued using that definition until about 1990. Then, together with my graduate student Theo Norvell, we had what we thought was a better idea, namely concurrency with merge. So in the first edition of [0] in 1993, concurrency includes merge. When Tony Hoare and He Jifeng wrote their book on *Unifying Theories of Programming* [2], which was published in 1998, they adopted my theory of programming, including concurrency with merge. But by then, I had 8 years programming experience with it, and I realized that the benefits in principle were not realized in practice. For programming by refinement, in practice, one has to partition the variables among the processes. In the meantime, Leslie Lamport introduced TLA [3], and in that theory concurrency is simple conjunction. From my experience and Leslie's urging, for the 2002 and later editions of [0], I returned to concurrency as simple conjunction.

Conclusion

Concurrency must be defined for all specifications, not just programs, in order to be able to program by refinement. And it must be monotonic in the refinement (implication) ordering. Concurrency as conjunction is simple, it describes activities that happen at the same time, it applies to all specifications, and it is monotonic. Adding a merge to form the final state strengthens the definition, but it adds complication, is more difficult to implement, and in practice does not help in programming.

Interleaving of atomic actions cannot be defined until we have a program, so interleaving is not a general definition of concurrency. But it is the only possible implementation when there are too few processors.

References

[0] E.C.R.Hehner: *a Practical Theory of Programming*, Springer, 1993; current edition hehner.ca/aPToP

[1] E.C.R.Hehner, C.A.R.Hoare: "a More Complete Model of Communicating Processes", *Theoretical Computer Science* v.26 p.105-120, North-Holland, 1983, published earlier as "Another Look at Communicating Processes", CSRG-TR134, University or Toronto, 1981

[2] C.A.R.Hoare, J.He: *Unifying Theories of Programming*, Prentice-Hall International, Series in Computer Science (ed. C.A.R.Hoare), London, 1998

[3] L.Lamport: the Temporal Logic of Actions, *ACM Transactions on Programming Languages and Systems* v.16 n. 3 p.872-923, 1994 May