Communicating Process Architectures 2007
All the editors (Eds.)
IOS Press, 2007

Fine-grain concurrency

Tony Hoare

Microsoft Research, Cambridge

Abstract. I have been interested in concurrent programming since about 1963, when its associated problems contributed to the failure of the largest software project that I have managed. When I moved to an academic career in 1968, I hoped that I could find a solution to the problems by my research. Quite quickly I decided to concentrate on coarse-grained concurrency, which does not allow concurrent processes to share main memory. The only interaction between processes is confined to explicit input and output commands. This simplification led eventually to the exploration of the theory of Communicating Sequential Processes.

Since joining Microsoft Research in 1999, I have plucked up courage at last to look at fine-grain concurrency, involving threads which interleave their access to main memory at the fine granularity of single instruction execution. By combining the merits of a number of different theories of concurrency, one can paint a relatively simple picture of a theory for the correct design of concurrent systems. Indeed, pictures are a great help in conveying the basic understanding. This paper presents some on-going directions of research that I have been pursuing with colleagues in Cambridge – both at Microsoft Research and in the University Computing Laboratory.

1. Introduction

Intel has announced that in future each standard computer chip will contain a hundred or more processors (cores), operating concurrently on the same shared memory. The speed of the individual processors will never be significantly faster than they are today. Continued increase in performance of hardware will therefore depend on the skill of programmers in exploiting the concurrency of this multi-core architecture. In addition, programmers will have to avoid increased risks of race conditions, non-determinism, deadlocks and livelocks. And they will have to avoid the usual overheads that concurrency libraries often impose on them today. History shows that these are challenges that programmers have found difficult to meet. Can good research, leading to good theory, and backed up by good programming tools, help us to discharge our new responsibility to maintain the validity of Moore's law?

To meet this challenge, there are a great many theories to choose from. They include automata theory, Petri nets, process algebra (many varieties), separation logic, critical regions and rely/guarantee conditions. The practicing programmer might well be disillusioned by the wide choice, and resolve to avoid theory completely, at least until the theorists have got their act together. So that is exactly what I propose to do. I have amalgamated ideas from all these well-known and well-researched and well-tested theories. I have applied them to the design of a structured calculus for low-overhead fine-grain concurrent programming. My theory of correctness is equally well-known: it is based on flowcharts and Floyd assertions. They provide a contractual basis for the compositional design and verification of concurrent algorithms and systems.

The ideas that I describe are intended to be an aid to effective thinking about concurrency, and to reliable planning of its exploitation. But it is possible to imagine a future in which the ideas can be more directly exploited. My intention is that a small collection of primitive operations will be simple enough for direct implementation in hardware, reducing

the familiar overheads of concurrency to the irreducible minimum. Furthermore, the correctness of the designs may be certified by future programming tools capable of verifying the assertions that specify correctness. And finally, the pictures that I draw may help in education of programmers to exploit concurrency with confidence, and so enable all users of computers to benefit from future increases in hardware performance. But I leave to you the judgement whether this is a likely outcome.

2. Sequential Processes, modeled by flowcharts

I will start with a review of the concept of a flowchart. It is a graph consisting of boxes connected by arrows. Each box contains basic actions and tests from the program. On its perimeter, the box offers a number of entry and exit ports. Each arrow connects an exit port of the box at its tail to an entry port of the box at its head.

Execution of the program is modelled by a control token that passes along the arrows and through the boxes of the flowchart. As it passes through each box, it executes the actions and tests inside the box. In a sequential program there is only one token, so entry of a token into a box strictly alternates with exit from the box. Furthermore, there is no risk of two tokens passing down the same arrow at the same time. We will preserve an analogue of this property when we introduce concurrency.

The example in Figure 1 shows the familiar features of a flowchart. The first box on the left has two exits and one entry; it is the purpose of the test within the box to determine which exit is taken by the token on each occasion of entry. The two arrows on the right of the picture fan in to the same head. After the token has passed through a fan-in, it is no longer known which of the two incoming arrows it has traversed.

As shown in Figure 2, the execution control token starts at a designated arrow of the flowchart, usually drawn at the top left corner of the diagram. We regard the token as carrying the current state of the computer. This includes the names and current values of all the internal variables of the program, as well as the state of parts of the real world that are directly connected to the computer. In this simple example, we assume the initial state on entry of the token ascribes the value 9 to x.

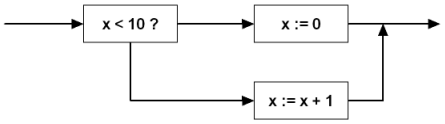

Figure 1. A flowchart

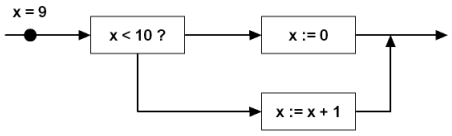

Figure 2. A flowchart with token – 1

As shown in Figure 3, execution of the test in the first box causes the token to exit on the lower port, without changing the value of x. In Figure 4, execution of the code in the next box increases the value of x by 1.

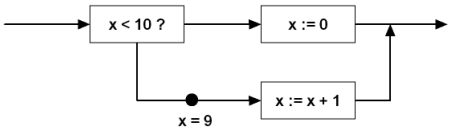

Figure 3. A flowchart with token – 2

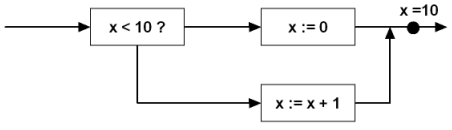

Figure 4. A flowchart with token – 3

In this sequence of diagrams, I have taken a snapshot of the passage of the token along each arrow. There is actually no storage of tokens on arrows, and conceptually, the emergence

of a token from the port at the tail of an arrow occurs at the same time as entry of the token into the port at the head of the arrow.

The previous figures showed an example of a conditional command, selecting between the execution of a **then** clause and an **else** clause. Figure 5 shows the general structure of a conditional command. It is general in the sense that the boxes are empty, and can be filled in any way you like. Notice that all the boxes now have one entry and two exits. The exit at the bottom of each of each box stands for the throw of an exception, implemented perhaps by a forward jump.

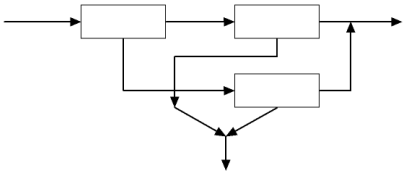

Figure 5. Conditional flowcharts

Figure 6 shows another useful generalisation of the concept of a flowchart, the structured flowchart: we allow any box to contain not only primitive commands of a program but also complete flowcharts. The pattern of containment must be properly nested, so the perimeters of different boxes do not intersect. Wherever an arrow crosses the perimeter between the interior and the outside of a containing box, it creates an entry or exit port, which is visible from the outside. Connections and internal boxes enclosed within the perimeter are regarded as externally invisible. Thus from the outside, the entire containing box can be regarded as a single command. The sole purpose of structuring is to permit flowcharts to be composed in a structured and modular fashion. The containing boxes are entirely ignored in execution.

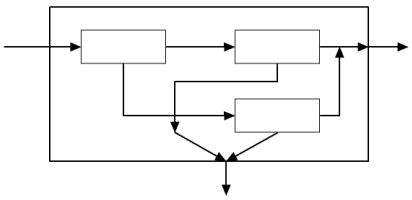

Figure 6. A structured flowchart

For convenience of verbal description, we will give conventional names to the entries and exits of each box as shown in Figure 7. The names are suggestive of the purpose of each port. In our simple calculus there will always be a start entry for initial entry of the token, a finish exit for normal termination, and a throw exit for exceptional termination. The names are regarded as local to the box. In pictures we will usually omit the names of the ports, and rely on the position of the arrow on the perimeter of the box to identify it.

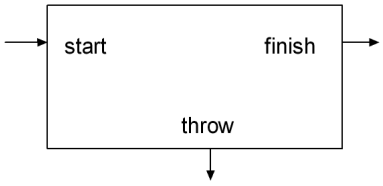

Figure 7. Port names

The ports of the enclosing boxes also have names. In fact, we generally use the same names for the enclosing box as well as the enclosed boxes. This is allowed, because port names inside boxes are treated as strictly local. The re-use of names emphasises the structural similarity of the enclosing box to the enclosed boxes. For example, in Figure 8, the enclosing box has the same structure and port names as each of the enclosed boxes. In fact, the whole purpose of the calculus that we develop is to preserve the same structure for all boxes, both large and small.

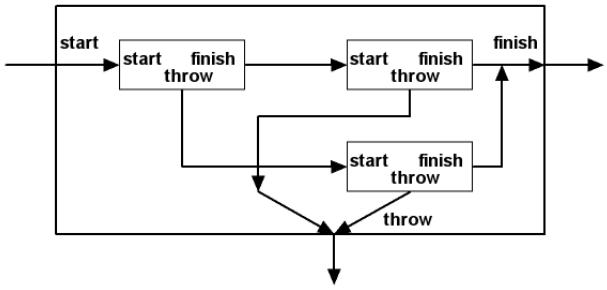

Figure 8. Structured naming

The notion of correctness of a flowchart is provided by Floyd assertions, placed on the entries and exits of the boxes. An assertion is a boolean condition that is expected to be true whenever a token passes through the port that it labels. An assertion on an entry port is a precondition of the box, and must be made true by the environment before the token arrives at that entry. The assertion on an exit port is a post-condition of the box, and the program in the box must make it true before sending the token out on that exit. That is the criterion of correctness of the box; and the proof of correctness is the responsibility of the designer of the program inside the box.

Figure 9 shows our familiar example of a flowchart, with assertions on some of the arrows. The starting precondition is that x is an odd number. After the first test has succeeded, its postcondition states that x is still odd and furthermore it is less than 10. After adding 1 to x, it is less than 11, and 1 more than an odd number. The postcondition of the other branch is obviously that x is 0. On both branches of the conditional, the postcondition on the extreme right of the flowchart states that x is even, and less than 11.

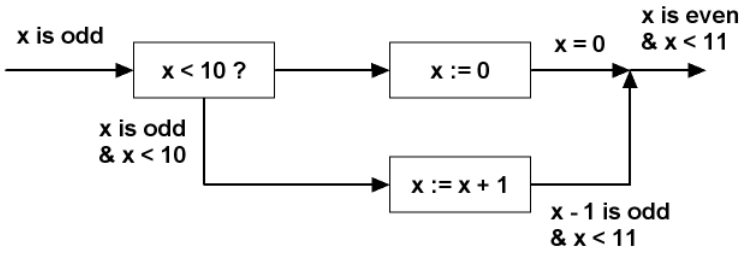

Figure 9. Flowchart with assertions

Let us examine the principles that have been used in this informal reasoning. The criterion of correctness for an arrow is very simple: the assertion at the tail of the arrow must logically imply the assertion at the head. And that is enough. As Floyd pointed out, a complete flowchart is correct if all its boxes and all its arrows are correct. This means that the total task of correctness proof of a complete system is modular, and can be discharged one arrow and one box at a time.

There is a great advantage in Floyd's method of formalising program correctness. The same flowchart is used both for an operational semantics, determining the path of the token when executed, and for an axiomatic semantics, determining the flow of implication in a correctness proof. There is no need to prove the consistency of the two presentations of semantics.

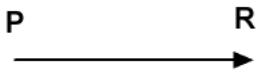

Figure 10. Arrow. The arrow is correct if $P \Rightarrow R$

We allow any number of arrows to be composed into arbitrary meshes. But we are not interested in the details of the internal construction of the mesh. We are only interested whether any given arrow tail on the extreme left has a connection path of arrows leading to a given arrow head on the extreme right. We ignore the details of the path that makes the connection. Two meshes are regarded as equal if they make all the same connections. So the mesh consisting of a fan-in followed by a fan-out is the same as a fully connected mesh, as shown in Figure 11. Wherever the mesh shows a connection, the assertion at the tail on the left must imply the assertion at the head on the right. The proof obligation can be abbreviated to a single implication, using disjunction of the antecedents and conjunction of the consequents.

Figure 11. Equal meshes. The mesh is correct if $P \vee Q \Rightarrow R \,\&\, S$

We will now proceed to give a definition of a little calculus of fine-grain concurrent programs. We start with some of the simplest possible boxes and flowcharts. The first example in Figure 12 is the simple skip action which does nothing. A token that enters at the start passes unchanged to the finish. The throw exit remains unconnected, with the result that it is never activated.

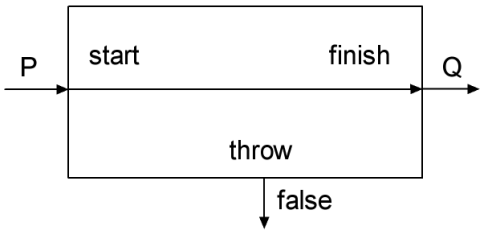

Figure 12. Skip action. The box is correct if $P \Rightarrow Q$

The proof obligation for skip follows directly from the correctness condition of the single arrow that it contains. The false postcondition on the throw exit indicates that this exit

will never be taken. Since false implies anything, an exit labelled by false may be correctly connected to any entry whatsoever.

The purpose of a throw is to deal with a situation in which successful completion is known to be impossible or inappropriate. The throw is usually invoked conditionally. Its definition is very similar to that of the skip, and so is its correctness condition. A flowchart for the throw action is shown in Figure 13.

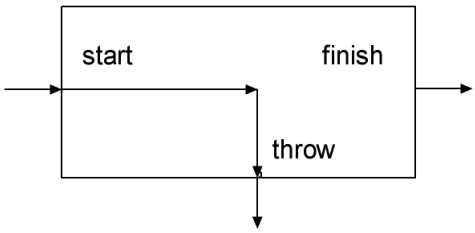

Figure 13. Throw action

The operators of our calculus show how smaller flowcharts can be connected to make larger flowcharts. Our first operator is sequential composition. We adopt the convention that the two operands of a composite flowchart are drawn as boxes inside an enclosing box that describes the whole of the composed transaction. The behaviour of the operator is determined solely by the internal connections between the ports of all three boxes. It is essential in a compositional calculus that the definition does not depend on the contents of its operand boxes. This rule is guaranteed if the internal boxes contain nothing, as shown in Figure 14.

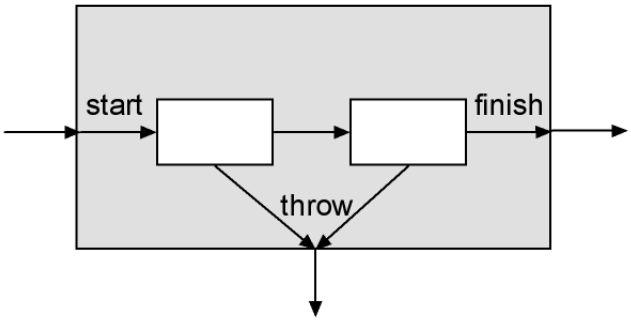

Figure 14. Sequential composition

To assist in proof of correctness, there should in principle be assertions on each of the arrows. However, the permitted patterns for these assertions are completely determined by the correctness principle for the arrows of a flowchart, so there is no need to mention them explicitly.

Sequential composition has many interesting and useful mathematical properties. For example, it is an associative operator. All the binary operators defined in the rest of this presentation will also be associative. Informal proofs of these and similar algebraic properties are quite simple. Just draw the flowcharts for each side of the equation, and then remove the boxes that indicate the bracketing. The two flowcharts will then be found to be identical. They therefore have identical executions and identical assertions, and identical correctness conditions.

Figure 15 shows the sequential composition of three transactions, with the gray box indicating that the brackets are placed to the left.

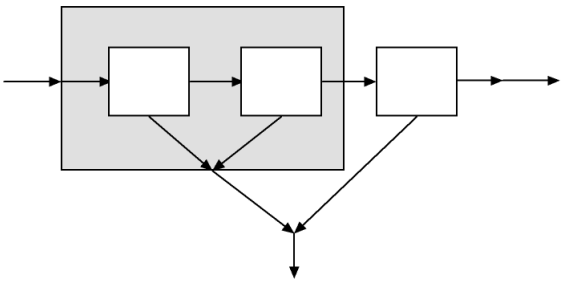

Figure 15. Asssociativity proof (left association)

And Figure 16 shows the same three processes with bracketing to the right. You can see that the flowcharts remain the same, even when the enclosing gray box moves. The apparent movement of the throw arrow is obviously not significant, according to our definition of equality of meshes of arrows.

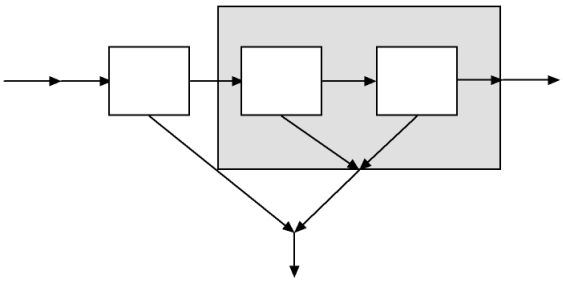

Figure 16. Asssociativity proof (right association)

In conventional flow-charts, it is prohibited for an arrow to fan out. Thus the thick arrow in Figure 17 would not be allowed. But we will allow fan-out, and use it to introduce non-determinism into our flowchart. When the token reaches a fan-out, it is not determined which choice it will make. This fact is exploited in the definition of a structured operator for non-deterministic choice between two operands. Whichever choice is made by the token on entry to the enclosing gray box, the subsequent behaviour of the program is wholly determined by the selected internal box. The other one will never even be started. The programmer must be prepared for both choices, and both must be correct. Non-determinism can only be used if the programmer genuinely does not care which choice is made. This is why non-determinism is not a useful operator for explicit use by programmers. We define it here merely as an aid to reasoning about the non-determinism that is inevitably introduced by fine-grain concurrency.

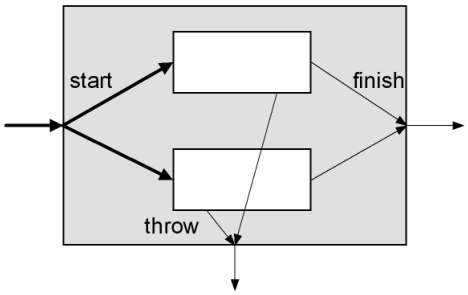

Figure 17. Non-determinism

Note that non-determinism is associative, but it has no unit. It is symmetric: the order in which the operands are written does not matter. It is idempotent: a choice between two identical boxes is the same as no choice at all. Finally, sequential composition, and most other forms of composition distribute, through nondeterminism. The proof of this uses Floyd's principle, that two flowcharts which have identical correctness conditions have the same meaning.

3. Concurrent Processes, modeled by Petri Nets

We now extend our notation for flowcharts to introduce concurrency. This is done by one of the basic primitives of a Petri net, the transition. As shown in Figure 18, a transition is drawn usually as a thick vertical bar, and it acts as a barrier to tokens passing through. It has entry ports on one side (usually on the left) and exit ports on the other. The transition transmits tokens only when there are tokens ready to pass on every one of its entry ports. These tokens are then replaced by tokens emerging simultaneously from every one of the exit ports. Note that transitions in themselves do not store tokens: the firing of a transition is an atomic event. We will later introduce Petri net places as primitive devices to perform the storage function.

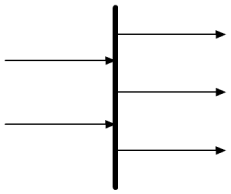

Figure 18. Petri net transition

As shown in Figure 19, if there is only one entry arrow, the transition is geometrically like a fan-out, since it contains two (or more) exit arrows. It is used to transmit a token simultaneously to a number of concurrent threads. It is therefore called a fork.

The other simple case of a transition is a join, as shown in Figure 20. It has only one exit port, and two or more entries. It requires tokens on all its inputs to pass through it simultaneously, and merges them into a single token. It thereby reduces the degree of concurrency in the system.

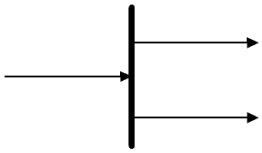

Figure 19. Petri net fork

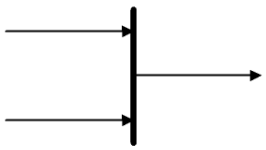

Figure 20. Petri net join

The simple cases of forks and joins are sufficient to reconstruct all the more complicated forms of a Petri net transition. This is done by connecting a number of transitions into a mesh, possibly together with other arrows fanning in and fanning out. A mesh with transitions is capable of absorbing a complete set of tokens on some subset of its entry arrows, delivering tokens simultaneously to some subset of its exit arrows. These two subsets are said to be connected by the mesh. In the case of a mesh without transitions, the connection is made between singleton subsets. Two general meshes are regarded as equal if they make exactly the same connections between subsets. So the mesh shown in Figure 21 is equal to the mesh shown in Figure 22.

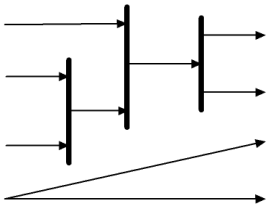

Figure 21. Petri net mesh – 1

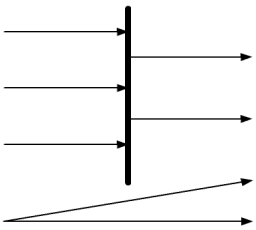

Figure 22. Petri net mesh – 2

An inappropriate mixture of transitions with fan-in and fan-out of arrows can lead to unfortunate effects. Figure 23 shows an example corner case. A token at the top left of the mesh can never move through the transition. This is because the fan-out delivers a token at only one of its two heads, whereas the transition requires a token at both of them. As a result, the whole mesh has exactly the same effect as a mesh which actually makes only one connection. We will design our calculus of concurrency to ensure that such corner cases will never arise.

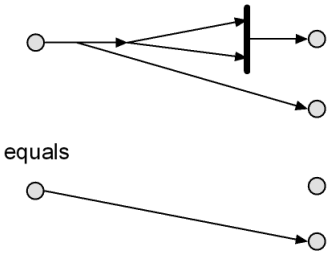

Figure 23. A corner case

In the design of fine-grain concurrent programs, it is essential to keep account of the ownership of resources by the threads which update them. We will therefore regard each token as carrying with it a claim to the ownership (i.e., the write permissions and read permissions) for just a part of the state of the computer; though for simplicity, we will largely ignore read permissions. Obviously, we will allow a box to access and update only the resources carried by the token that has entered the box. The addition of ownership claims to the tokens helps us to use Petri nets for their initial purpose, the modelling of data flow as well as control flow through the system.

In Figure 24, the ownership of variables `x` and `y` is indicated by writing these names on the token which carries the variables. Figure 25 is the state after firing the transition. The resources claimed by the token are split into two or more disjoint parts (possibly sharing read-only variables); these parts are carried by the separate tokens emerging from the fork.

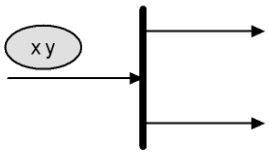

Figure 24. Token split: before

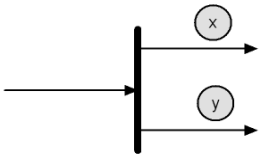

Figure 25. Token split: after

In Figure 24 and Figure 25, token at entry carries whole state: {`x y`}; at the exits, each sub-token carries a disjoint part of the state.

The Petri net join is entirely symmetric to the fork. Just as the fork splits the ownership claims of the incoming token, the join merges the claims into a single token. In Figure 26 and

Figure 27, each sub-token carries part of the state at entry; at exit, the token carries whole state again.

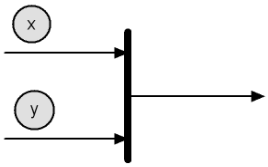

Figure 26. Token merge: before

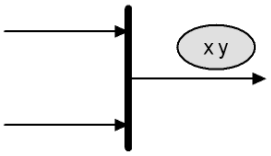

Figure 27. Token merge: after

What happens if the incoming tokens make incompatible claims on the same resource? Fortunately, in our structured calculus this cannot happen. The only way of generating tokens with different ownership claims is by the fork, which can only generate tokens with disjoint ownership claims. As a result, the claims of each distinct token in the entire system are disjoint with the claims of all the others. The join transition shown above preserves this disjointness property. So no resource is ever shared between two distinct tokens.

We allow the assertion on an arrow of a Petri net to describe the ownership claims of the token that passes along the arrow. For simplicity, we will just assume that any variable mentioned in the assertion is part of this claim. In reasoning with these assertions, it is convenient to use a recently introduced extension of classical logic, known as separation logic; it deals with assertions that make ownership claims.

Separation logic introduces a new associative operator, the separated conjunction of two predicates, usually denoted by a star ($P \star Q$). This asserts that both the predicates are true, and furthermore, that their ownership claims are disjoint, in the sense that there is no variable in common between the assertions. The ownership claim of the separated conjunction is the union of the claims of its two operands.

In a program that uses only declared variables without aliasing, the disjointness of the claims can be checked by a compiler, and separation logic is not necessary. The great strength of separation logic is that it deals equally well with pointers to objects in the heap. It allows any form of aliasing, and deals with the consequences by formal proof. However, our example will not illustrate this power of separation logic.

The axiom of assignment in separation logic is designed to prevent race conditions in a fine-grain concurrent program. It enforces the rule that the precondition and the postcondition must have the same claim; furthermore, the claim must include a write permission for the variable assigned, and a read permission for every variable read in the expression that delivers the assigned value. In the displayed axiom of assignment (Figure 28) we have exploited the common convention that a proposition implicitly claims all variables that it mentions. So the precondition and postcondition claim x and y. Because of disjointness, R must not claim x or y. For simplicity, we have failed to distinguish read and write permissions.

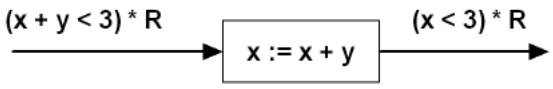

Figure 28. Axiom of assignment

Separated conjunction is used to express the correctness condition for Petri net transitions. The assertion at the entry of a must imply the separated conjunction of all the assertions at the exits. In Figure 29, the disjointness of P and Q represents the fact that the outgoing tokens will have disjoint claims.

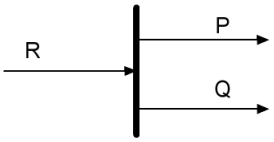

Figure 29. Correctness condition of fork: $R \Rightarrow P \star Q$

As mentioned before, the join is a mirror image of the fork. Accordingly, the correctness condition for a join is the mirror image of the correctness condition for a fork.

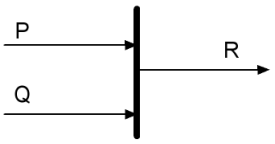

Figure 30. Correctness condition for join: $P \star Q \Rightarrow R$

There is a problem here. What happens if $P \star Q$ is false, even though both P and Q are both true? This would mean that the execution of the program has to make falsity true when it fires. But no implementation can do that – it is a logical impossibility. Fortunately, the rule of assignment ensures that P and Q must be consistent with each other. The details of the consistency proof of separation logic are beyond the scope of this paper.

The first example of the use of transitions in our calculus is the definition of the kind of structured (fork/join) concurrency introduced by Dijkstra. In Figure 31, the fork on the left ensures that both the threads labelled T and U will start together. The join on the right ensures that they will finish together. In between these transitions, each of the threads has its own token, and can therefore execute concurrently with the other. By definition of the fork and join, the tokens have disjoint claims. Since a thread can only mention variables owned by its token, the rule of assignment excludes the possibility of race conditions. It also excludes the possibility of any interaction whatsoever between the two threads.

In Figure 31, I have not allowed any possibility of a throw. The omission will be rectified shortly.

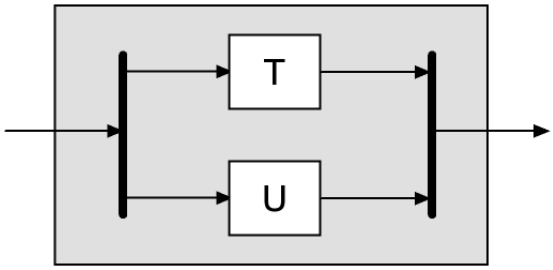

Figure 31. Concurrent composition. There is no connection between T and U

Figure 32 is a simple example of a concurrent program. The precondition says that x and y have the same parity. One thread adds 2 to x, and the other multiplies y by 7. Both these operations preserve parity. So the same precondition still holds as a postcondition. Although this is obvious, the proof requires a construction, as shown in Figure 33. The construction introduces an abstract or ghost variable z to stand for the parity of x and y. A ghost variable may appear only in assertions, so it remains constant throughout its scope. For the same

reason, a ghost variable can be validly shared among threads (though it may not be either read or written). When it has served its purpose, the ghost variable may be eliminated by existential quantification in both the precondition and the postcondition.

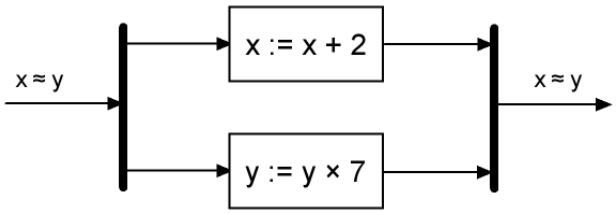

Figure 32. A concurrent composition example. $x \approx y$ means (x - y) mod 2 = 0 (their difference is even)

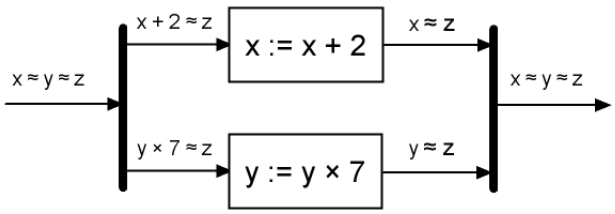

Figure 33. Ghost variable z

Proof:
$$x \approx y \;\Rightarrow\; x + 2 \approx y \times 7$$
$$x \approx y \approx z \;\Rightarrow\; (x + 2 \approx z) \star (y \times 7 \approx z)$$

We now return to the example of the structured concurrency operator and remove the restriction on throws. In Figure 34, the throw exits of T and U are connected through a new join transition to the throw exit of the composition. As a result, the concurrent combination throws just when both the operands throw. This still leaves an unfortunate situation when one of the operands attempts to throw, whereas the other one finishes normally. In an implementation, this would manifest itself as a deadlock.

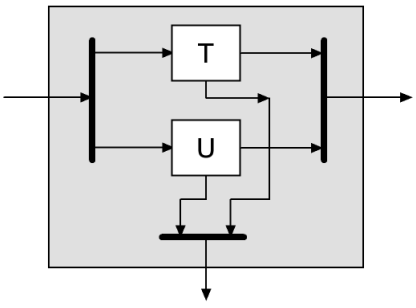

Figure 34. Concurrency with throw. To avoid deadlock, T and U must agree on their exits

A solution is to adopt an even more complicated definition of concurrent composition. It ensures that a throw will occur when either of the operands throws, even if the other one finishes. As shown in Figure 35, this is achieved by additional joins to cover the two cases when the threads disagree on their choice of exit port.

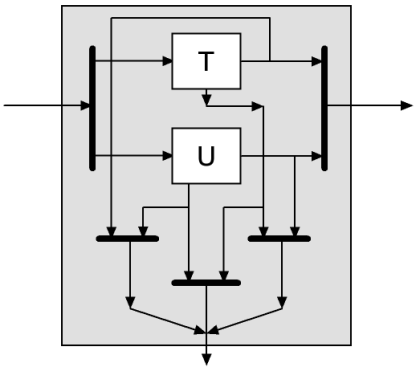

Figure 35. Deadlock avoided. Disagreement on exit leads to throw

In Figure 36, note the four encircled fan-outs in the arrows at the exits of the operands T and U. Each of these introduces non-determinism. However, it is non-determinism of the external kind that is studied in process algebras like CCS and CSP. It is called external, because the choice between the alternatives is made at the head of the arrow rather than at the tail. On reaching the fan-out, the token will choose a branch leading to a transition that is ready to fire, and not to a transition that cannot fire. In Figure 36, we have ensured that at most one of the alternative transitions can be ready to fire. Thus the diagram is in fact still completely deterministic, in spite of the four fan-outs.

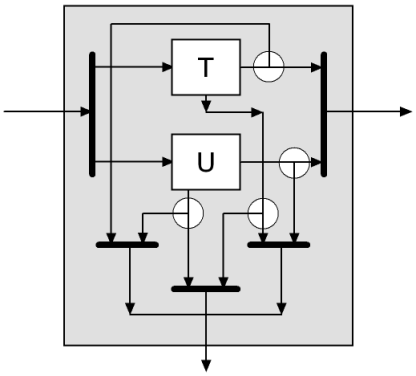

Figure 36. Fan-out gives external non-determinism

The calculus that we have described so far is not capable of exploiting fully the power of multi-core architecture. The reason is that the same rules that prohibit race conditions also prohibit any form of communication or co-operation among the threads. To relax this restriction, it is necessary to establish some method of internal communication from one thread to another. For the purpose of exploiting multi-core architecture, the highest bandwidth, the minimum overhead and the lowest latency are simultaneously achieved by use of the resources of the shared memory for communication. Communication takes place when one thread updates a variable that is later read by another.

Of course, race conditions must still be avoided. This is done by the mechanism of a critical region, which enables the programmer to define a suitable level of granularity for the interleaving of operations on the shared resource by all the sharing threads. A critical region starts by acquiring the shared resource and ends by releasing it, through new entry ports introduced into our calculus for this purpose. Inside a critical region, a thread may freely update the shared resource together with the variables that it owns permanently. Race conditions are still avoided, because the implementation ensures that at any time at most one

thread can be in possession of the critical region. A simple implementation technique like an exclusion semaphore can ensure this.

In our Petri net model, a shared resource is represented by a token which carries ownership of the resource. In order to access and update the shared resource, a thread must acquire this token, which is done by means of a standard join between the control token and a token carrying ownership of the resource. After updating the shared state within the critical region, the thread must release the token, by means of a standard fork. The standard rules of ownership are exactly appropriate for checking critical regions defined in this way, since the token that travels through the region will carry with it the ownership of both the local variables of the thread and the variables of the shared resource. These can therefore be freely updated together within the critical region.

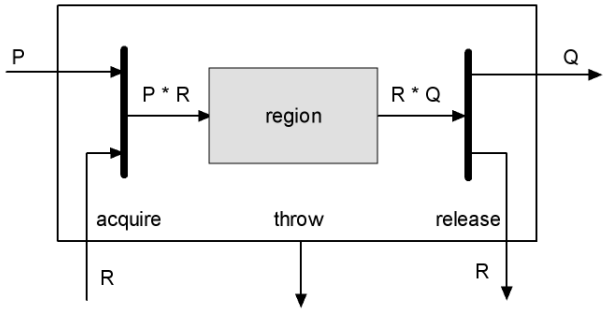

Figure 37. Critical region. R is the resource invariant

Note that the body of the critical region has no acquire or release ports. This intentionally prohibits the nesting of critical regions. Furthermore, I have disallowed throws from within a critical region. To allow throws, the definition of a critical region requires an additional fork transition to ensure that the resource token is released before the throw exit. This means that the programmer must restore the resource invariant before the throw.

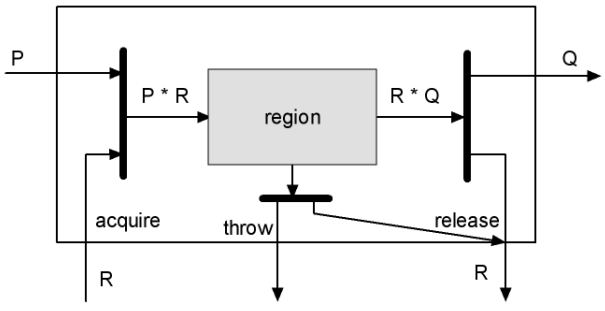

Figure 38. Critical region with throw

Addition of new ports into a calculus requires extension of the definition of all the previously defined operators. In the case of the new acquire and release ports, the resource is equally accessible to all the operands, and the standard extension rule is to just connect each new entry port of the enclosing block for the operator by a fan-out to the like-named new entry ports of both the operands; and connect every new exit port of each operand via a fan-in to the like-named port on the enclosing block. Figure 39 shows only the new ports and additional arrows that are to be added to every operator defined so far. It ensures that the new ports can be used at any time by either of the operands.

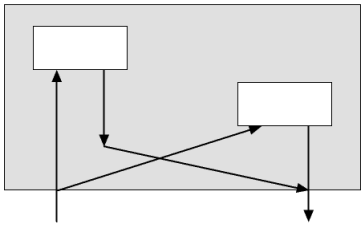

Figure 39. New ports

A shared resource is introduced by exactly the same operator which introduces multiple threads. The token that owns the resource is created by the fork on the left of Figure 40. It then resides at a place (denoted by a circle) specially designated for it within the Petri net. The resource token is acquired by its users one at a time through the acquire entry at the beginning of each critical region, and it is released after use through the release exit at the end of each critical region. It then returns to its designated place. If more than one user is simultaneously ready to acquire the resource token, the choice between them is arbitrary; it has to be made by the semaphore mechanism that implements exclusion. This is the way that shared memory introduces don't-care non-determinism into a concurrent program.

The assertion R in this diagram stands for the resource invariant. As shown in Figure 39, it may be assumed true at the beginning of every critical region, and must be proved true at the end. It thus serves the same role as a guarantee condition in the rely/guarantee method of proving concurrent programs.

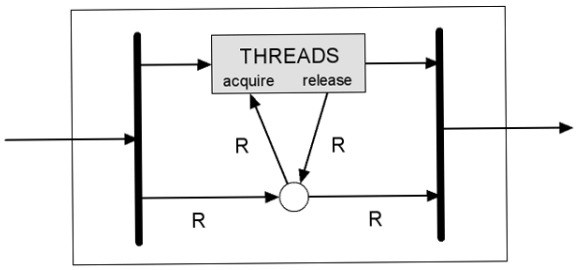

Figure 40. Resource declaration. Petri net place: ◯ stores a token

Figure 41 caters for the possibility of a throw, in the usual way.

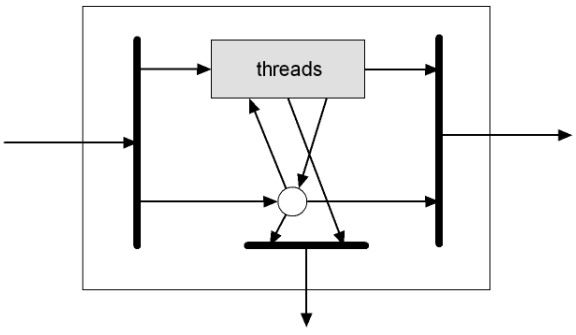

Figure 41. Resource declaration with throw

Figure 42 is an extremely simple example of concurrency with critical regions. Two threads share a variable x. One of them assigns to it the value 2, and the other one assigns the

value 7. Because the variable is shared, this has to be done in a critical region Each thread is nothing but a single critical region. As a result, the two critical regions are executed in arbitrary order, and the final value of x will be either 2 or 7. The easiest proof is operational: just prove the postcondition separately for each of the two interleavings. But in general, the number of interleavings is astronomical. So we want to ask whether our assertional proof system capable of proving this directly in a more abstract way?

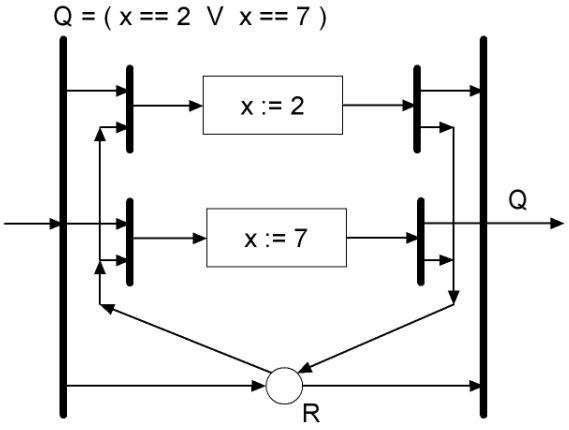

Figure 42. Example – 1

The answer seems to be yes, but only with the help of a ghost variable t, introduced to record the termination of one of the threads. The variable obviously starts false. By conditioning the resource invariant on t, its truth is assured at the beginning. Both critical regions leave the resource invariant R true. And one of them sets t true. Thus at the end, both t and R are true. Thus Q is also true at the end.

But the question arises, who owns t? It has to be joint ownership by the resource and the first thread. Such jointly owned variables can be updated only in a critical region, and only by the thread that half-owns it. The resource owns the other half. When the resource and the thread have come together in the critical region, full ownership enables the variable to be updated. This is adequate protection against race conditions. Fractional ownership is a mechanism also used for read-only variables in recent versions of separation logic.

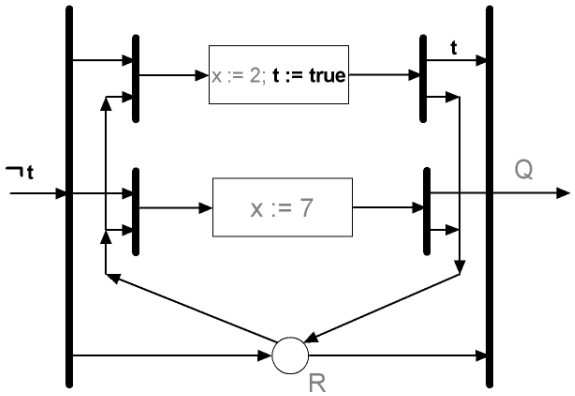

Figure 43. Example – 2. $Q = x \in \{2, 7\}$ and $R = t \Rightarrow Q$

4. Other features of a calculus

Recursion is the most important feature of any programming calculus, because it allows the execution of a program to be longer than the program itself. Iteration is of course an especially efficient special case of recursion. Fortunately, Dana Scott showed how to introduce recursion into flowcharts a long time ago. Just give a name X to a box, and use the same name as the content of one or more of the interior boxes. This effectively defines an infinite net, with a copy of the whole box inserted into the inner box. For this reason, the pattern of entry and exit ports of the recursive call must be the same as that of the outer named box. That is a constraint that is easily enforced by use of a calculus like one we have described.

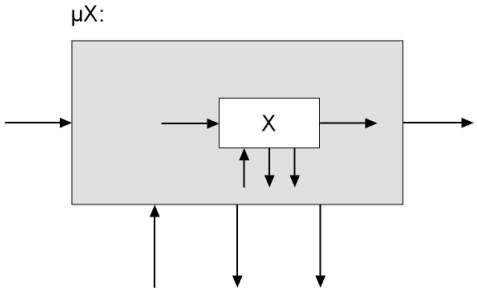

Figure 44. Scott recursion

A variable can be represented by a place pre-loaded with a token that owns the variable. This token joins the main control token on entry to the block, which can use the variable as required. It is forked off again on exit from the block, so that it is never seen from the outside. A place is needed at the finish to store the token after use. Let us use the same place as stored the token at the beginning.

The assertions on the arrow leading from and to the place should just be the proposition true, which is always true. This means that nothing is known of the value of the variable immediately after declaration. It also means that its value on termination is irrelevant. This permits an implementation to delay allocation of storage to the variable until the block is entered, and to recover the storage or exit.

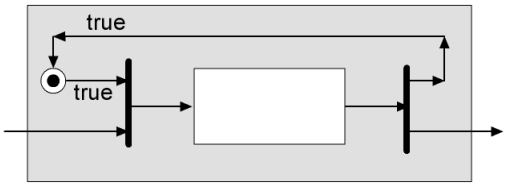

Figure 45. Variable declaration – 1

Figure 46 extends the diagram to show what happens on a throw. The variable still needs to be retained inside the box after an exception.

The Petri net fork is a direct implementation of an output from one thread of a system to another. It simply transfers ownership of the message (together with its value) to the inputting process. It does not copy the value. It does not allocate any buffer. Overhead is therefore held to a minimum. If buffers are desired, they can be modelled as a sequence of Petri net places.

Just as output was a fork, input is a join at the other end of an arrow between two threads. Note that the output is synchronised with the inputting process. In a sympathetic architecture (like that of the transputer), the operations of input and output can be built into the instruction set of the computer, thereby avoiding software overhead altogether.

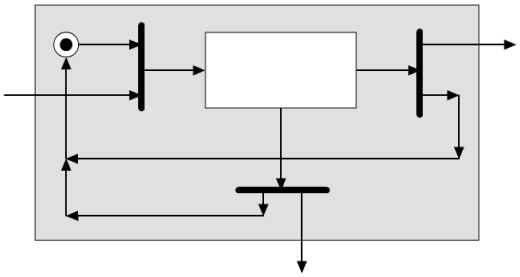

Figure 46. Variable declaration – 2

The introduction of arbitrary arrows communicating ownership among threads can easily lead to deadlock. Absence of deadlock can be proved by the methods of process algebra, and we will not treat it here. Fortunately, the use of non-nested critical regions is a disciplined form of communication which is not subject to deadlock. A simple hierarchy of regions can extend the guarantee to nested regions.

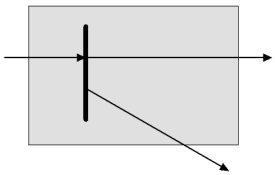

Figure 47. Output

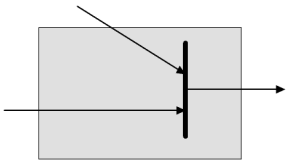

Figure 48. Input

5. Conclusion

The main conclusions that may be drawn from this study are:

1. Flow-charts are an excellent pictorial way of defining the operational semantics of program components with multiple entry and exit points. Of course, they are not recommended for actual presentation of non-trivial programs.
2. Floyd assertions are an excellent way of defining and proving correctness of flowcharts. Consistency with an operational semantics for flowcharts is immediate.
3. Petri nets with transitions extend these benefits to fine-grain concurrent programs. The tokens are envisaged as carrying ownership of system resources, and permissions for their use.
4. Separation logic provides appropriate concepts for annotating the transitions of a Petri net. The axiom of assignment provides proof of absence of race conditions.
5. Critical regions (possibly conditional) provide a relatively safe way of using shared memory for communication and co-operation among threads.
6. Although they are not treated in this paper, rely/guarantee conditions provide a useful abstraction for the interleaving of critical regions.
7. Pictures are an excellent medium for defining the operators of a calculus. They are readily understood by programmers who are unfamiliar with programming language semantics (some of them even have an aversion to syntax).

Of course, there is abundant evidence, accumulated over many years, of the value of each of these ideas used separately. The only novel suggestion of this presentation is that their combined use may be of yet further value in meeting the new challenges of multi-core architecture.

Acknowledgment

Thanks to Robert Floyd, Carl Adam Petri, Cliff Jones, Simon Peyton Jones, Tim Harris, Viktor Vafeiadis, Matthew Parkinson, Wolfgang Reisig and Steve Schneider. Even though there are no references, it is a pleasure to express my thanks to those who have inspired this work, or helped its progress.