Available online at www.sciencedirect.com
ScienceDirect
Electronic Notes in Theoretical Computer Science
ELSEVIER
Electronic Notes in Theoretical Computer Science 209 (2008) 165–171
www.elsevier.com/locate/entcs

Verification of Fine-grain Concurrent Programs

Tony Hoare[1]

Microsoft Research Ltd.
Cambridge CB3 0FB.

Abstract

Intel has announced that in future each standard computer chip will contain many processors operating concurrently on the same shared memory; their use of memory is interleaved at the fine granularity of individual memory accesses. The speed of the individual processors will never be significantly faster than they are today. Continued increase in performance 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. To reduce these risks, we propose a theory of correctness for fine-grain concurrent programs. The approach is just an amalgamation of a number of well-known and well-researched ideas, including flowcharts, Floyd assertions, Petri nets, process algebra, separation logic, critical regions and rely/guarantee reasoning. These ideas are applied in combination to the design of a structured calculus of correctness for fine-grain concurrent programs; it includes the main features of a structured concurrent programming language.

Keywords: Fine-grain concurrent programs, Flowcharts, Floyd assertions, Petri nets

1 Semantic foundation

We construct a model of a program as a Petri net, which generalises the familiar flowchart by introducing data flow and concurrency. The boxes of the net contain the basic actions (assignments, communications, jumps and tests) of the program. They have entry and exit ports, connected by arrows to the ports of other boxes. For convenience, the ports are given standard names like *start*, *finish* and *throw*. The execution of a Petri net is modelled by the passage of tokens along the arrows, executing the program code in the boxes as they pass through. Execution of the program inside each box determines the exit port through which the token leaves the box.

The notion of program correctness is provided by Floyd assertions, placed on the ports of the boxes; an assertion is expected to be true whenever a token passes

[1] Email: thoare@microsoft.com

doi:10.1016/j.entcs.2008.04.010

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. The assertion on an exit port is a post-condition of the box, and the program in the box must make it true before exit. Obviously, when ports are connected by an arrow, correctness requires that any assertion on an exit port at the tail of an arrow must imply every assertion on the entry ports at the head of that arrow. This simple rule can be checked by purely local reasoning. That is how a meaning is assigned to the whole program represented by the net; in fact, the assertional correctness-oriented semantics for Petri nets follows directly from the operational semantics that prescribes the movement of the token in the net.

The more dynamic aspects of the operational semantics can be specified by a notional trace of the order in which the token passes through the various named entry and exit ports of each box in the net. For example, there are only two possible traces of a normal sequential program: they are *start;finish* and *start;throw*. This small set is generated by the regular expression: start ; (*finish* + *throw*). In fact, all the boxes of a normal sequential program follow this pattern. For boxes which have additional entry and exit ports, useful classes of program can be defined by specifying the pattern of their traces. This provides a simple method of protecting against the dangers of deadlock.

2 Modularity

A modular structure can be imposed on a Petri net by enclosing certain of its subnets into a box, with its own named ports on the perimeter. The assertions on these ports describe what that box assumes from its neighbours, and what it guarantees. Thus for purposes of reasoning about correctness, the entire enclosing box can be regarded as a single action, ignoring the details of shape and content of the subnet which it contains. In execution, however, the enclosing boxes are completely ignored.

The dynamic pattern of behaviour of an enclosing box is often intended to be the same as that of the boxes that it encloses. To make this explicit, it is a useful convention to hide (e.g., erase) the port names of the internal boxes, and re-use them as port names for the enclosing box. Thus the same regular expression can then be re-used to describe the dynamic pattern of both the enclosing and the enclosed boxes. This convention is exploited in the definition of a small calculus of operators by which Petri nets can be composed in a well structured fashion. An operator of the calculus is defined in terms of a box that encloses its operands. Some of the ports of the operands are connected to each other, and other internal ports are connected to ports on the perimeter of the enclosing box. The pattern of connections defines the meaning of the operator. A proof is then given that the enclosing box follows the same pattern of dynamic behaviour as its operands.

As a simple example, consider the operation of conventional sequential composition. Let us name its operands P and Q , and name the enclosing box (P;Q) . Inside the enclosing box, the *start* entry of (P;Q) is connected to the *start* entry

of P . The *finish* exit of P is connected to the *start* entry of Q . The *finish* exit of Q is connected to the *finish* exit of (P;Q) . These three connections ensure the correct flow of normal control between the two operands. The *throw* exits for both P and Q are connected to the *throw* exit for (P;Q) . This indicates that a throw from either operand results in a throw from their composition. From the outside, it is not known which operand has triggered the throw. This pattern of connections is more easily described by a picture than in words.

This definition of sequential composition can easily be proved healthy, in the sense that it preserves the standard pattern of behaviour for sequential programs. If the two operands observe the pattern, so does the enclosing box. Other important properties of sequential composition such as associativity can easily be established from the fact that the enclosing box is ignored in execution. Similar treatment can be given to the definition of a try-catch operator and a conditional operator. The latter has three operands, and uses the throw exit of the first operand to select between the other two.

3 Concurrency

Concurrency is introduced, controlled, and eliminated in a Petri net by means of transitions. The transition is a primitive component of a net, drawn usually as a bar, and it acts as a barrier to tokens passing through. It has entry ports on one side and exit ports on the other. The transition transmits tokens only when there are tokens waiting on every one of its entry ports. These tokens are then replaced by tokens emerging simultaneously from every one of the exit ports. If there is only one entry port, the transition acts as a fan-out, transmitting control to many concurrent threads. If there is only one exit port, the token acts as a fan-in, synchronising the termination of the threads. It helps to describe these cases separately. A more general transition can easily be treated as a fan-out followed by a fan-in.

Transitions can be used to define an operator for the kind of structured concurrency introduced by Dijkstra. Let us denote the concurrent threads by P and Q , and their concurrent combination by (P——Q) . The *start* entry for (P——Q) is connected to the entry of a binary fan-out. One exit from the fan-out is connected to the *start* entry of P , and the other to the *start* entry of Q . This means that the two threads start together. The *finish* exit of P is connected to one of the entries of a binary fan-in. The *finish* exit of Q is connected to the other entry of this fan-in. The exit of the fan-in is connected to the *finish* exit of (P——Q) . This means that the two threads also finish together. The *throw* exits of P and Q are similarly connected through a fan-in transition to the throw exit of (P——Q) . This means that the concurrent combination throws just when both the operands throw. Actually, we want the throw to occur when either of the operands throws. This can be achieved by additional fan-ins to cover each case when the threads disagree on their choice of exit port.

In the design of concurrent programs, it is almost essential to keep account of the ownership of the resources of the computer by the threads. We will regard

each token as carrying with it the ownership (i.e., write permissions) for just a part of the state of the computer, and the channels which connect it to the real world environment. We will call them resources. A token also carries a record of partial ownership of a resource, (i.e., read permissions). When the token passes through a box, the program in the box is allowed to access and update only those parts of the overall state whose partial or total ownership is carried by the token. This restriction could be implemented physically by a hardware memory in which every variable carries a permission tag. But it is far more practical to prove in advance that a program observes all the relevant restrictions.

When a token reaches a fan-out transition, the resources that it owns are split into two or more disjoint parts (possibly sharing read-only variables); these parts are carried by the separate tokens emerging from the fan-out. When a token reaches a fan-in transition, it is merged into a single token, together with a token from each of the other entry ports of that fan-in. The token that emerges carries the sum of all the permissions of all the tokens that pass through the barrier at the same time. This sum must be consistent, in the sense no write permission is added to any other permission, read or write.

Fortunately, observance of this rule is guaranteed by the fact that any two tokens in the system always carry disjoint permissions. This is because the only way of generating tokens is by a consistent split of existing tokens. If a language which allows dynamic generation of resources, a new resource is always disjoint from all existing ones. As a result, at all times the entire resources of the system are split disjointly among all the tokens travelling in the net. This ensures that the ownership claims of two tokens arriving at the same fan-in transition will always be compatible.

4 Separation logic

Reasoning about ownership is conveniently conducted in the framework of a new variety of modal logic, known as separation logic. The distinctive feature of separation logic is that each assertion implicitly (or explicitly) makes an ownership claim on the variables that it mentions or depends on. The logic is effective in dealing with heap variables, whose ownership cannot generally be determined by compile time checks. The logic introduces a new associative operator, the separated conjunction of two predicates, usually denoted by a star (P*Q) . This asserts that both the predicates are true, and furthermore, that their ownership claims are disjoint. Separated conjunction is used to express the correctness condition for Petri net transitions. The assertion at the entry of a fan-out is just the separated conjunction of all the assertions at the exits. Conversely in the case of a fan-in, the assertion on the exit is the separated conjunction of all the assertions on the entries.

We have to worry about the possibility that this conjunction on fan-in is inconsistent (equal to false). A program that satisfies a false postcondition can be proved to satisfy any specification whatsoever. Such a program cannot be implemented. Any calculus which permits proof of a false postcondition is significantly flawed, and its use must be restricted to some partial or conditional form of correctness.

It is highly desirable to design a calculus that avoids false postconditions. This is done by ensuring that any construction that could lead to a false postcondition is given a false precondition. This is safe, because the programmer has to prove that the precondition is true before the construction is entered, and this cannot be done if the precondition is false.

Inconsistency on fan-in arises from race conditions. For example, if two threads make incompatible assignments to the same variable, then an inconsistency arises on the fan-in which implements their synchronised termination. Fortunately, in separation logic assignment to the same variable by different threads is ruled out. The primitive axiom of assignment checks that ownership of the assigned variable is carried by the token, and also checks the necessary read permissions to evaluate the expression. If the check fails, it is the precondition that is set false. Thus the kind of race condition that leads to inconsistency is ruled out by the proof of correctness of the net.

5 Shared memory

Unfortunately, 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 communication between threads. For the purpose of exploiting multi-core architecture, the highest bandwidth, minimum overhead and lowest latency are achieved by use of shared memory for communication. Of course, sharing must be confined to a region of memory disjoint from the memory that is permanently owned by the threads that share it. Ownership of the shared resource is carried in the normal way by a token, which resides normally at a place specially assigned to it within the Petri net. This token must therefore be split off by a fan-out before the start of the threads that are going to share it; and it should be merged again after all these threads have terminated. The declaration of a shared resource can be achieved by the same concurrency operator as that described above.

In order to access and update the shared resource, a thread must acquire its token by means of a fan-in transition. After updating the shared state, the thread must release the token back to its designated place. The region of the thread program which enjoys temporary ownership of a shared token is called an atomic or critical region. Use of this kind of critical region gives the programmer control over the granularity of the interactions of the threads, at a level above that of individual memory accesses.

The same technique extends to a larger collection of independent resources, which can be acquired and released separately. If more than one resource is required at the same time by the same thread, care is needed to avoid deadlock. A useful pattern of usage is always to acquire the resources in a standard order. Release of the token does not have to be so tightly controlled, provided that the prover can accurately match each acquisition to its release.

When several processes are simultaneously ready to acquire the token for a

shared resource, the one that gets it is selected non-deterministically. As a result, the order of execution of the critical regions is an arbitrary interleaving of the critical regions of all the threads that share the resource. To reason effectively about this non-determinism, a useful level of abstraction is provided by rely conditions and guarantee conditions. Both of these are relations between two states of the shared resource. The guarantee condition for a thread is a relation that holds between the initial state and the final state of the resource at the beginning and the end of every critical region within that thread. The rely condition for a thread is a relation which holds between the final state at the end of each critical region and the initial state at the beginning of the next critical region executed (where initiation and termination of the thread are treated as the end and beginning of a critical region). The rely condition plays the role of an assumption in the proof of each thread. It places an upper bound on the changes to the shared resource which the other threads can make while the given thread does not own it. The concurrent execution of all threads sharing a resource is correct if each threads guarantee condition implies the rely condition of all the other threads. In the case of multiple resources, there can be a separate rely and guarantee condition for each pairing of each thread with each resource.

6 Conclusion

The slideshow that accompanied presentation of this note gave pictorial definitions of the operators of a small calculus of concurrency, including the main conceptual features of a concurrent programming language, with proof rules for correctness. It seemed possible to convey a useable understanding of the calculus without the aid of formal syntax or semantics.

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

(i) Flow-charts are an excellent pictorial way of representing program components with multiple entry and exit points. Of course, they are not recommended for actual presentation of non-trivial programs.

(ii) Floyd assertions are an excellent way of defining and proving correctness of flowcharts. Consistency with an operational semantics for flowcharts is immediate.

(iii) 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.

(iv) Separation logic provides appropriate concepts for annotating the transitions of a Petri net. The axiom of assignment provides proof of absence of race conditions.

(v) Critical regions (possibly conditional) provide a relatively safe way of using shared memory for communication and co-operation among threads.

(vi) Rely/guarantee conditions provide a useful abstraction for the interleaving of critical regions.

(vii) Pictures are an excellent medium for defining the operators of a calculus. They are also accessible 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 exploiting multi-core architecture.