Top-Down Considerations on Distributed Computing*

Ron van der Meyden[1] and Yoram Moses[2]

[1] University of Technology Sydney, P.O.Box 123 Broadway, NSW 2007 Australia
[2] Faculty of Electrical Engineering, The Technion, Haifa 3200 Israel

Distributed programs and systems are notoriously complex to design and prove correct. Even very short and seemingly intuitive distributed programs have been shown to behave in unexpected ways. As a result, the literature on distributed algorithms has tended to concentrate on solving specific problems in isolation. While this is an effective way of dealing with the complexity of distributed computing, the solutions obtained for manageable subproblems must eventually be composed to form more complex systems.

We are currently engaged in a research project concerned with developing a methodology for the top-down design of distributed programs. The purpose of the talk is to discuss some issues related to compositionality that arise in this work, and that may be relevant more broadly. This abstract presents in an informal manner a few of the points that will be discussed in the talk.

Sequential Composition: A very common practice in problem solving is to *divide and conquer*: Break a problem into simpler subproblems, solve each of them, and combine the solutions to a solution of the original problem. We could, for example, want to design a protocol for voting by breaking it into three parts: (a) Compute a minimum spanning tree (MST) of the network; (b) elect a leader using the MST; and (c) coordinate the vote through the leader. Clearly, these operations should be performed in sequence. Solutions to the abovementioned subproblems are typically assumed to start operating at a well-defined initial global state. But solutions to the MST problem or to leader election are not guaranteed to terminate at a well-defined global state [2]. How, then, do we compose the solutions? It is possible to overcome this problem by performing a synchronization step at the end of the intermediate steps, say in the form of termination detection. The cost of synchronizing the processes in the network can be high, however, and should be incurred only if needed. In our particular example this should not be necessary.

Extending the intuitions underlying the work on communication-closed layers [3, 1, 6, 7], we prefer to view distributed programs as operating between *cuts*, where a cut specifies an instant on the time-line of each of the processes. Intuitively this means that, as far as sequential composition is concerned, cuts constitute a distributed analogue of states of a sequential computation. With this view, it is possible to sequentially compose terminating programs such as those in our voting example.

* this work is supported by a grant from the Australian Research Council

The seemingly small move of viewing distributed programs as operating between cuts has considerable repercussions. For example, we now need to consider much more carefully what assumptions must be made about the *initial cut* of a given program, to ensure the program behaves in the desirable fashion. In some cases, the initial cut needs to be fairly tightly synchronized, perhaps a global state or something very close to it such as a consistent cut. In other cases, we can make do with much less. For example, in many cases it suffices that a message that is sent by process i in the course of executing its portion of the distributed program P and is received by j before j is ready to execute its part of P, will be presented again to j after j starts executing its portion of P. Properly formalized, such a condition is all that is needed to solve our voting example in a reliable asynchronous model, without needing to perform costly synchronization between the three layers of the computation. A characterization of distributed problems and solutions according to the degree of synchronization they require and those they provide is an important open problem.

High-level programs: It is possible to extend the discussion of properties and assumptions on initial cuts one step further. Informally, let us define a *cut formula* to be a formula that is interpreted over cuts. For every cut formula φ there will be a set of pairs r, c where r is a run and c is a cut, at which φ will be considered true. Cut formulas can be treated just like state formulas in a sequential computation. In fact, they enable the definition of branching and iteration at the level of distributed programs. If P and Q are distributed programs and φ is a cut formula, we can define

$$\textbf{if } \varphi \textbf{ then } P \textbf{ else } Q \qquad \text{and} \qquad \textbf{while } \varphi \textbf{ do } P$$

as *high-level* distributed programs. High-level programs of this nature resemble structured programs in sequential computing. But there is an important difference between the two. Whereas in the sequential case the tests in **if** and **while** statements are usually executable and can be accepted by the compiler as such, the corresponding tests on cut formulas are not directly executable in general. Nevertheless, such high-level constructs of distributed programming can serve as rigorous *descriptions* of the desired behavior. This, in turn, can be helpful in the process of designing a distributed program, when the designer wishes to transform a specification of a desired program into a concrete implementation. For example, we might decide to implement a program for computing the MST of a network in an incremental fashion by repeatedly adding an MST edge to the forest under construction. This will correspond to executing a loop of the form **while** φ **do** P, where φ is the cut formula stating that the MST is not completed yet, and P is a distributed program that adds one edge to the MST.

Termination: We have argued that considering programs as operating between cuts provides us with an improved facility for handling sequential composition of distributed programs. We need to assume, of course, that each participant in a distributed program ultimately terminates its participation in the program, before it can go on to performing the next program. This is taken for granted when dealing with sequential programs, where a nonterminating computation amounts to an outright failure and is usually totally undesirable. When we deal with distributed programs, there are common settings in which it is provable that desirable tasks cannot be solved by terminating distributed programs. For example, the work of Koo and Toueg [5] shows that in a setting in which messages may be lost but communication channels are *fair*, every protocol that requires the successful delivery of at least one message in each run, must have executions in which one or more of the processes cannot terminate. This appears to be an issue if we are after sequential composition of programs in this model.

An example of a nonterminating program in this context is the standard protocol for sending a single bit between a sender S and a receiver R. The sender's program is to send the bit repeatedly until it receives an acknowledgement. The receiver, in turn, sends one acknowledgement for each message it receives. A close analysis shows that the receiver must forever be ready to send one more acknowledgement in case it receives another copy of the bit. This example indicates that while the receiver cannot safely reach a terminating state with respect to the bit transmission program, the situation is not similar to the divergence of a sequential program. Despite not reaching a terminating state, the receiver should not be expected to abstain from taking part in further activities, provided that the receipt of a new copy of the bit will prompt the sending of an additional acknowledgement.

Inspired by the work of Havelund and Larsen [4], we subscribe to an approach by which part of the activity of a process can be considered as taking place "in the background." We call a program that operates in the background a *forked* program, and have an explicit $\mathbf{fork}(P)$ operator to capture this in our framework. The distinction between foreground and background activities has to do with how the compose with later programs. A forked activity takes place concurrent with whatever comes next, it need not terminate, and and if it does, its termination need not cause another activity to start. In the bit transmission problem, for example, we can view the receiver having a *top-level* or "foreground" activity consisting of waiting for the sender's message, and consuming it, or making use of the information therein once it arrives. The "background" activity, which the receiver would happily delegate to an assistant or, say, the mail system, involves sending the acknowledgements. Once the message is obtained by the receiver, the receiver can proceed to its next task. A similar distinction between aspects of distributed programs can be applied to a variety of well-known protocols and problems.

A separation of concerns between foreground and background activities allows the designer of a program to make fine-grained distinctions in specifying the control structure of the program. It enables mixing terminating programs with nonterminating programs and makes sequential composition of programs possible again even in models or for problems where termination is not a simple matter.

In summary, we feel that the concerns about composition of distributed programs raised above should receive more attention than they have in the past. We believe that they could give rise to new problems and techniques in the field of distributed algorithms. In our work, the observations made in this abstract form the basis of a refinement calculus for the top-down development of distributed programs from specifications. A detailed description of that work will be the subject of future papers.

References

1. Chou, C., Gafni, E.: Understanding and verifying distributed algorithms using stratified decomposition. Proc. 7th ACM PODC (1988) 44–65
2. Gallager, R., Humblet, P., Spira, P.: A distributed algorithm for minimum-weight spanning trees. ACM Trans. on Prog. Lang. and Syst., **5**(1) (1983) 66–77
3. Elrad, T., Francez, N.: Decomposition of distributed programs into communication-closed layers. Sci. Comp. Prog., **2**(3) (1982) 155-173
4. Havelund, K., Larsen, K.G.: The fork calculus. Proc. 20th ICALP, LNCS **700** (1993) 544–557
5. Koo, R., Toueg, S.: Effects of message loss on termination of distributed protocols. Inf. Proc, Letters, **27** (1988) 181–188
6. Stomp, F., de Roever, W.P.: A principle for sequential reasoning about distributed systems. Form. Asp. Comp., **6**(6) (1994) 716–737
7. Zweirs, J., Janssen, W.: Partial-order based design of concurrent systems. Proc. REX Symp. "A decade of concurrency", J. de Bakker, W. P. de Roever, G. Rozenberg eds., LNCS **803** (1994) 622-684