Information Processing Letters 21 (1985) 1–6
North-Holland 10 July 1985

BIDIRECTIONAL COROUTINES *

Hanan SAMET
Department of Computer Science, University of Maryland, College Park, MD 20742, U.S.A.

Communicated by Alan C. Shaw
Received 29 December 1983
Revised 27 September 1984, 28 January 1985

The bidirectional coroutine is introduced as a mechanism for overcoming a shortcoming in the method of specification of the transfer of control between coroutines. An analogy is drawn between subroutines and coroutines by observing that coroutines, like subroutines, should not have to know with whom they are interacting. At present, most coroutine implementations require specific mention of the coroutine being resumed, or use a **suspend** mechanism in which case one coroutine acts as a slave (the suspending one) and the other as a master. In the second case, the slave need not know the identity of its master while the master must know the identity of its slave. For bidirectional coroutines, a coroutine need not know the identity of its master nor its slave. This is achieved by replacing the **suspend** primitive with two new primitives — **resume_master** and **resume_slave**.

Keywords: Coroutines, semicoroutines, hierarchical coroutines

1. Introduction

The coroutine [2,1] concept is important in the design of control structures for programming languages. In a taxonomy of control structures it fits somewhere between a subroutine and parallel processing (i.e., tasking). It is useful when it is necessary to model a situation where two or more processes must operate in a handshaking manner —i.e., process A does a bit of work, at which time it relinquishes control to process B which also does some work and subsequently returns control to process A, process A continues for awhile at which time it resumes process B at the point where B last returned control to A. This pattern of sharing of control is different from a subroutine because a subroutine can only be invoked at its start. It is less general than parallel processing because each coroutine depends on the actions of other coroutines whereas in the case of parallel processing the processes could be executed autonomously.

In this article we point out a shortcoming in the way in which the transfer of control between coroutines is specified. In particular, it is our view that just as subroutines need not know who invoked them, the same feature should be available when using coroutines. At present, most coroutine implementations either require specific mention of the coroutine being resumed (termed a *symmetric* coroutine [8]), or use a suspend mechanism (termed a *semi-symmetric* [8] or hierarchical [7] coroutine). In the latter case, one coroutine acts as a slave (the suspending one) and the other acts as a master. The slave need not know the identity of its master but the master must know the identity of its slave. At times, it may also be convenient for the master not to know the identity of its slave. In order to achieve this symmetry, we develop the concept of a bidirectional coroutine. Our approach is an iterative one starting with a simple **goto**, refining it to yield a subroutine, symmetric coroutine, hierarchical coroutine and finally a bidirectional coroutine.

* This research has been supported in part by the National Science Foundation under Grant DCR-8302118.

0020-0190/85/$3.30 © 1985, Elsevier Science Publishers B.V. (North-Holland)

2. Bidirectional coroutines

The simplest programming construct for transferring control is the **goto**. It has fallen into disfavor because there is no structure associated with it. If the programmer wishes to return to the site of the **goto**, then he must explicitly remember it. The subroutine is a control structure which resolves such problems. It permits the creation of separate programs which can be invoked by other programs. The invoked program, termed a *subroutine* (i.e., a procedure or a function), need not know who invoked it in order to return control upon completion to its invoker (also termed its *caller*). A call to a subroutine, say A, is like an interrupt and A executes to completion unless A calls another subroutine. Thus there is a nested flow of control (i.e., stack-like). As an example, consider Fig. 1 where we show the interaction between three procedures, A, B, and C. Note that A calls B which eventually calls C. When C is through, B is resumed and it runs to completion at which time A is resumed.

The coroutine is an attempt to make the interrupt characterization of the subroutine symmetric. Recall that, for a subroutine, the caller is resumed where the interrupt occurred. Once the called subroutine has returned control to its caller, it cannot be resumed again except at its start. Using coroutines it is possible for two programs to interact with each other in a handshaking manner. For example, consider Fig. 2 where we show the coroutine interaction between two coroutines, A and B. Such an interaction is termed *symmetric* [8]. Notice that we no longer have a flow of control that can be managed by a single stack. If there are only two coroutines, then once one coroutine has initiated the other (i.e., established a coroutine-like link), say A has initiated B, then we can simply use the primitive **resume** to denote a switch in control. If there are more than two coroutines, then we must specifically name the coroutine that is being resumed (i.e., **resume**(A)). It is useful to observe that there does not exist a hierarchical relationship between the coroutines. Each coroutine invokes another as if it were the main program.

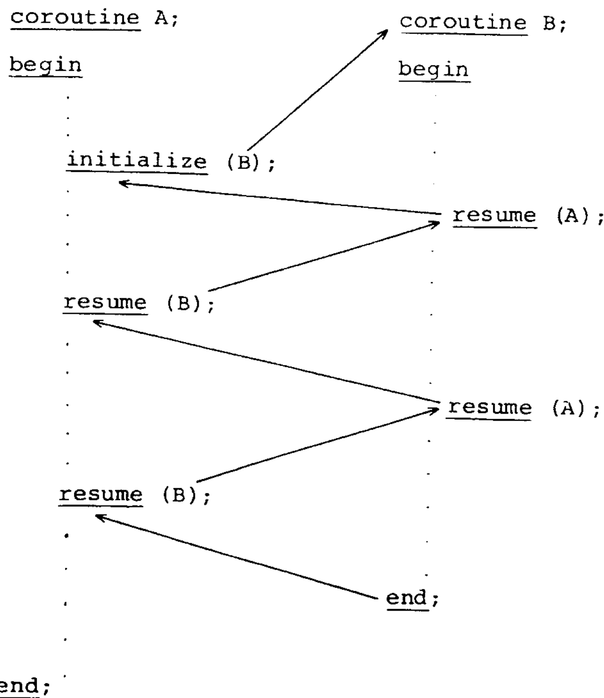

Fig. 2. Example coroutine interaction.

The interaction between two subroutines can be characterized as a master–slave relationship where the caller is the master and the called subroutine is the slave. A similar relationship can also be established between two coroutines. In essence, the initiating coroutine acts as a master and the initiated coroutine acts as a slave. Often, the subordinate coroutine acts as a producer and the

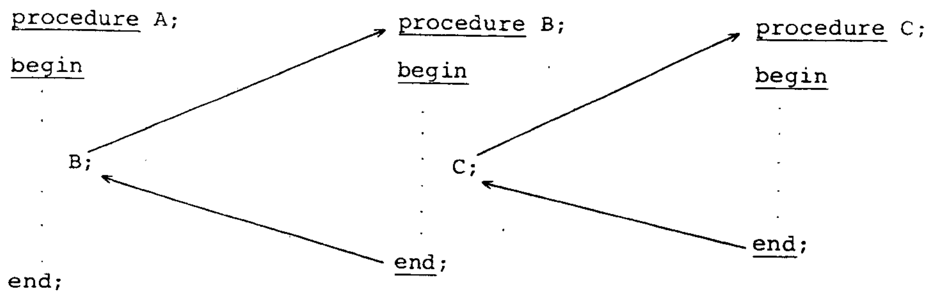

Fig. 1. Example subroutine interaction. (*Note*: Underlined parts of Figs. 1–4 are shown boldface in text.)

master acts as a consumer (or vice versa). The slave need not know the identity of its master. It returns control to its master by use of a **suspend** operation which is a version of **resume** without explicitly naming the successor. In contrast, the master must know the identity of its slave since it can have more than one slave. Thus, the master resumes the appropriate slave, say B, via use of the primitive **resume** (i.e., **resume**(B)). Note that a slave may also be a master of another coroutine. The flow of control is hierarchical in the sense that when a coroutine terminates—i.e., it exits via execution of its final statement or executes a **return** as in a subroutine—then we assume that control is returned to its master. All remaining coroutines which are slaves of the terminating coroutine are implicitly terminated. Coroutines that interact in such a manner are termed *semi-symmetric* by Wang and Dahl [8] and *hierarchical* by Vanek [7]. Dahl and Hoare [3] use the term *call/detach* to describe the master–slave relationship between two or more coroutines. They also call the slave coroutine a *semicoroutine* and use the primitive **detach** in an equivalent manner to **suspend**.

In his formulation of the hierarchical coroutine, Vanek stipulates that the master does not need to know whether the routine that is being called is a coroutine or not. Thus, he does not distinguish between initial coroutine invocation and coroutine resumption. As an example, consider Fig. 3, where

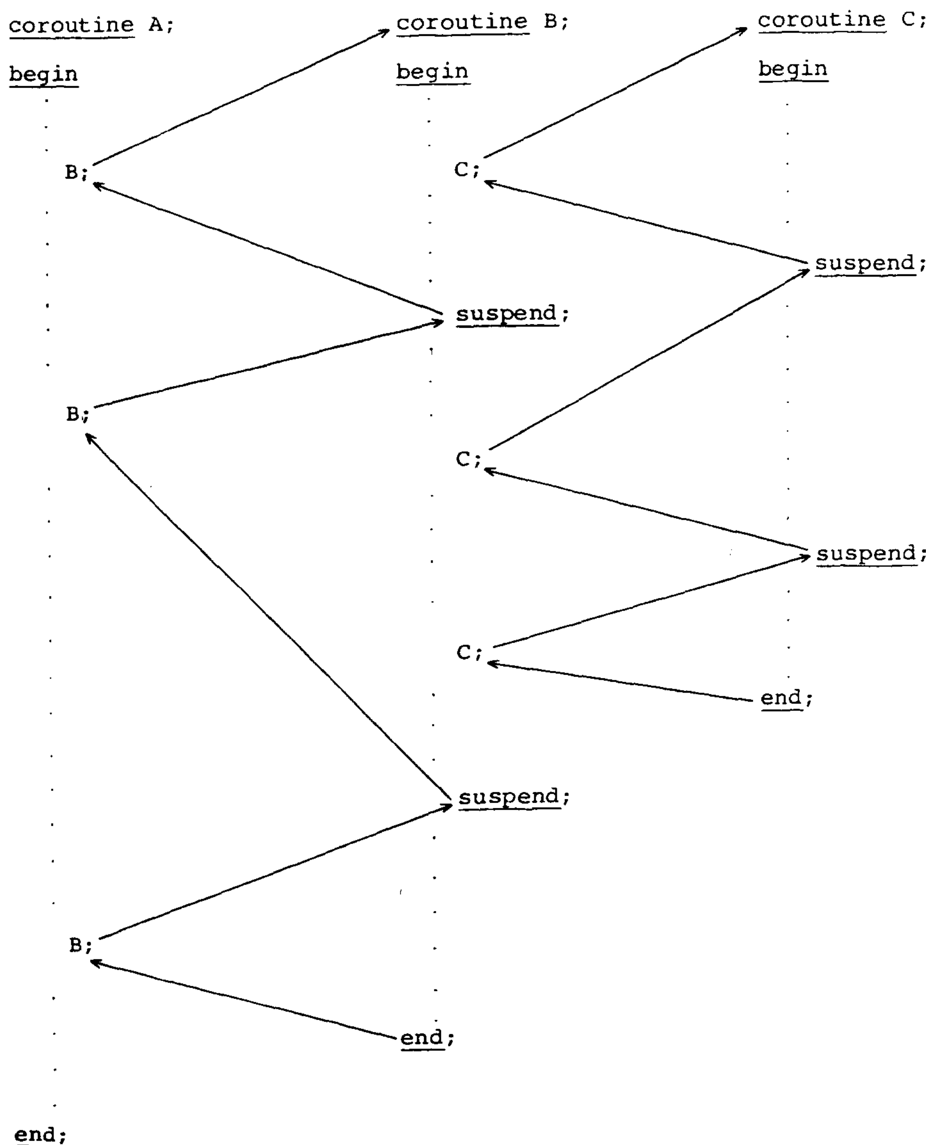

Fig. 3. Example hierarchical coroutine interaction.

we show the interaction between coroutines A, B, and C. Note that A is a master of B. B is both a slave of A and a master of C. If B should terminate while A and C are still active, then C would be implicitly terminated. Vanek uses static nesting to indicate coroutine dependence.

Symmetric coroutines have the shortcoming that each coroutine must know its successor or predecessor (e.g., a parser). Because they lack structure enforcing constructs, symmetric coroutines permit the creation of convoluted code fragments which may lead to a degree of confusion similar to that achieved by the **goto**. A partial remedy is provided by hierarchical coroutines. We are now at the main point of this article. As we saw above, the master–slave relationship of hierarchical coroutines is somewhat one-sided in the sense that in the event of coroutine resumption the slave need not know the identity of its master whereas the master must know the identity of the slave that it is resuming. Clearly, when a master has several slaves, then it must be able to distinguish among them. However, we adopt the hierarchical principle that when a master suspends without explicitly specifying which slave is to be resumed, and if there are several slaves, then the most recently suspended slave is resumed. Since a coroutine can be both a master and a slave simultaneously we

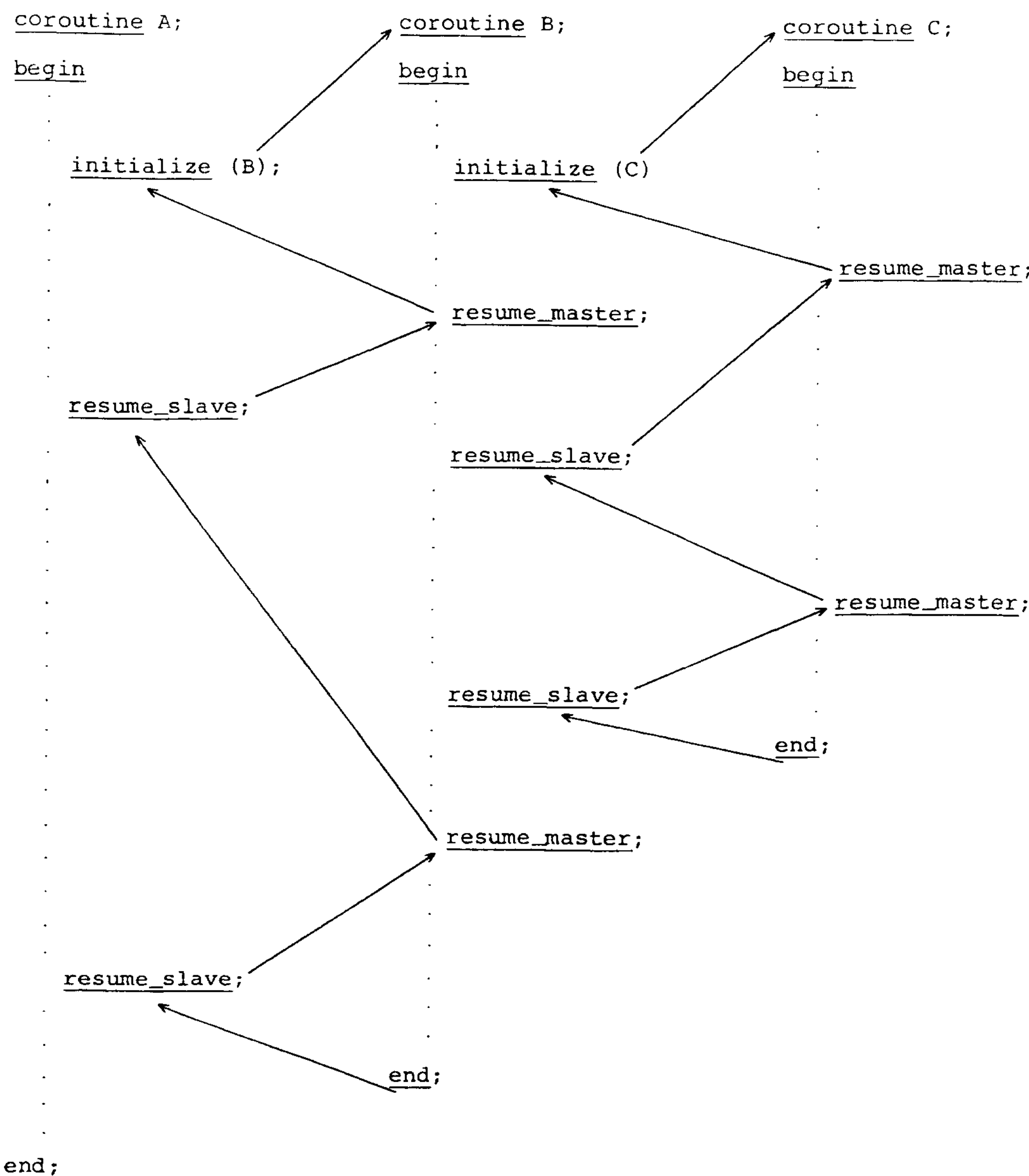

Fig. 4. Example bidirectional coroutine interaction.

introduce a new resume primitive **resume_slave** corresponding to the situation that a master resumes its slave and rename the **suspend** primitive to be **resume_master**. Note that **resume_slave** must still be used in conjunction with a **resume**(x) operation. As an example, consider Fig. 4 which is identical to Fig. 3 with the exception that once again we distinguish between initial coroutine invocation and coroutine resumption, and we use the **resume_master** and **resume_slave** primitives. The initial interaction is between A and B. Once the interaction between B and C is started, when we are in B we have a choice of resuming A (via **resume_master**) or resuming C (via **resume_slave**).

When coroutines are permitted to interact in the manner described above, we say that they are *bidirectional*. We use the term bidirectional because each coroutine can act like a subroutine with respect to the coroutine with which it is interacting. It need only know if it is a slave or a master with respect to the other coroutine. As an example of the utility of bidirectional coroutines, suppose that we replace the invocation of coroutine C in Fig. 4 by the following program fragment which tests a condition and depending on its value initiates coroutine WRITER or coroutine READER. All subsequent interactions can be done by use of **resume_slave** without having to test the condition again. Here we see an example where the target of **resume_slave** is ambiguous. Of course, an alternative approach is to use variables of type **coroutine** (e.g., **environment** in SL5 [4]) to denote the coroutine that is being resumed. Nevertheless, we prefer our approach for the same reason that **return** and **suspend** are used instead of return address variables for subroutines and semicoroutines.

```
if ⟨cond⟩ then coroutine_initiate(WRITER)
else coroutine_initiate(READER);
  ⋮
resume_slave;
  ⋮
```

Bidirectional coroutines are also useful in a simulation environment where tasks do not always have to know who invoked them in order to transfer control among themselves. Alternatively, a consumer need not always know the identity of his supplier (i.e., producer).

Notice that our use of the **resume_slave** primitive does not comport with Vanek's stipulation that the master should not distinguish between initial coroutine invocation and coroutine resumption. However, this uniformity is only applicable for the transfer of control from a master to a slave. Recall that control is transferred from a slave to a master in a different way (i.e., by use of the **suspend** primitive). In contrast, our approach treats the resumption of a master and a slave in a uniform manner at the cost of requiring a distinction between initial coroutine invocation and coroutine resumption.

3. Concluding remarks

We have developed the concept of a bidirectional coroutine in an iterative manner starting with the simplest of control structures. In this article, our goal was simply to introduce this concept and to motivate it. It should be clear that much more work remains in defining the full ramifications of its introduction into a programming language. The **resume_master** primitive can be implemented by Wang and Dahl's **swap** while the **resume_slave** primitive would require a variation of **swap** which takes as its argument the special element P representing the execution environment. In addition, we must define more carefully the accessibility of variables in such a context and binding rules upon coroutine resumption. For a good exposition on the semantics of coroutines and interaction among variables of different coroutines, see ACL of Marlin [6]. See also the discussion by Lindstrom and Soffa [5] on referencing and retention in the context of hierarchical coroutines.

Bidirectional coroutines can be used in an operating system or a simulation environment where the scheduler is both a master and a slave. The scheduler could serve as a slave to some processes. The occurrence of certain conditions would cause the scheduler to invoke other subprocesses as a master. Based on the response from these sub-

processes, the scheduler would either resume them or its invoking process. In general, examples are difficult to construct in a vacuum. An evaluation of the actual utility of bidirectional coroutines awaits their implementation and use.

Acknowledgment

I have benefitted from discussions with Fredrick Boland, John Gannon, Glenn Pearson, Olaf Schoenrich, and Deepak Sherlekar. I would like to thank Simon Kasif for suggesting the name bidirectional.

References

[1] G.M. Birtwistle, O.J. Dahl, B. Myhrhaug and K. Nygaard, SIMULA Begin (Auerbach Publishers Inc., Philadelphia, PA, 1973).

[2] M.E. Conway, Design of a separable transition-diagram compiler, Comm. ACM 6 (7) (1963) 396–408.

[3] O.J. Dahl and C.A.R. Hoare, Hierarchical program structures, in: O.J. Dahl, E.W. Dykstra and C.A.R. Hoare, eds., Structured Programming (Academic Press, London, 1972) 184–193.

[4] D.R. Hanson and R.E. Griswold, The SL5 procedure mechanism, Comm. ACM 21 (5) (1978) 392–400.

[5] G. Lindstrom and M.L. Soffa, Referencing and retention in block-structured coroutines, ACM Trans. on Programm. Languages and Systems 3 (3) (1981) 263–292.

[6] C.D. Marlin, Coroutines, Lecture Notes in Computer Science 95 (Springer, Berlin, 1980).

[7] L. Vanek, Hierarchical coroutines: A mechanism for improved program structure, Tech. Rept. No. 99, Computer Systems Research Group, University of Toronto, 1979.

[8] A. Wang and O.J. Dahl, Coroutine sequencing in a block structured environment, BIT 11 (1971) 425–449.