THE SAFETY OF A COROUTINE SYSTEM *

Jorma Sajaniemi
Department of Computer Science, University of Helsinki
Töölönkatu 11, SF-00100 Helsinki 10, Finland

ABSTRACT: Coroutine control is usually represented in high level languages in a way which allows the referencing of terminated coroutines and permits tricky calling sequences. In this paper a different method and notation for the expression of coroutines is outlined and its safety and balance are proved. The proofs are based on a model which describes only the relevant and general aspects of executions. Thus the proofs deal with the whole language and not with specific programs.

1. INTRODUCTION

Coroutines were first introduced over a decade ago [2] in the context of machine coding. The basic idea was to interweave execution of routines that otherwise should have been done one after another. The benefit gained by coroutines lay primarily in the avoidance of large files carrying information from one routine to the next one as these files could be transmitted piecemeal during the execution of the whole system of coroutines. In the above case information flowed only in one direction and so a natural extension was to allow free transmission of data. This turned coroutines into the form known today: they give a general way to interweave the execution of routines depending on each other.

When adopted to higher level languages [1, 3, 4, 5, 8] coroutines were generalized to a powerful control structure but the demand for simplicity and security was often forgotten. For example, it became possible to reference terminated coroutines and variables declared in them. The first kind of references must be considered to be errors while references to variables can be allowed if the memory areas of coroutines are not destroyed during termination.

In the following, we will outline a method and notation for recursive coroutines. The system is based on the grouping of co-operating coroutines [5]. These coroutines may have parameters that can be renewed by their colleagues during execution. In order to maintain the balance of groups of coroutines, parameter renewals are restricted by

* This work was supported by the Academy of Finland.

compile-time rules. We then show that the system has certain properties: control is always passed between coroutines of the same group, terminated coroutines and variables declared in them are never referenced, and the dynamic ancestors of a coroutine cannot change.

The proofs are based on a model similar to the one which is given in [7] and which we call the WD-model. Essentially, our model can be considered to be obtained from the WD-model by imposing some restrictions. Consequently our model has new properties as indicated in the above paragraph.

2. OUTLINE OF THE COROUTINE SYSTEM

In order to present the important aspects of the used version of coroutine control we give as an example the following sketch of a group of three coroutines. The group reads characters according to some convention (e.g. "5A" means "AAAAA"), outputs it in some readable form, and builds a nested data structure (e.g. a tree). The group consists of three coroutines, one for each of these tasks. As the tasks may be recursive it is convenient to use coroutines that may contain recursive procedures. This is shown in the third coroutine. The control flows through these coroutines in the cycle build-readchar-writechar-build-...

```
cogroup readdata;
begin
    coroutine readchar;
        begin character ch;  ...  writechar(ch)  ...  end;
    coroutine writechar(c); value c; character c;
        begin  ...  build(c)  ...  end;
    coroutine build(c); value c; character c;
        begin
            procedure substructure;
                begin  ...  readchar  ...  substructure  ...  end;
            substructure
        end;
    start build()
end
```

The declaration of a group, hereafter called cogroup, contains thus the coroutines and a start notice giving the starting coroutine. Normal static scope rules make it impossible to reference coroutines outside the cogroup but they can be referenced in nested cogroups. The model of section 3 will make the meaning of such references clear.

A cogroup can be invoked by its identifier in the block where it is defined or in the inner blocks. Its execution starts by the activation of the starting coroutine which can then pass control to other coroutines. These will be entered at their first calls. At later calls a coroutine continues its execution from the point where it was when it last called another coroutine. When one of the coroutines reaches its end the whole cogroup is terminated.

Coroutines may have parameters which are handled almost as in the case of procedures. The main exception is that these can be renewed during later calls. When a parameter is renewed, its old value is replaced by a new one. If some parameter is not renewed, its old value will be retained. There is also a restriction concerning parameters passed to coroutines: all identifiers in actual parameters passed by name to coroutines must be global to the called coroutine. We shall see that this restriction ensures the symmetry between co-operating coroutines.

Coroutines and cogroups are handled in other respects just like procedures. Thus coroutines and cogroups, as well as procedures, can be passed as parameters to coroutines, cogroups, and procedures.

3. THE MODEL

We will now introduce a model for the execution of programs written in the coroutine system outlined in section 2. It should be noted that we are not interested in specific programs but rather in the class of all programs. Thus we can leave out all details concerning the program to be executed if we instead give axioms that are satisfied by all programs. For example, the model does not keep track of identifiers but only states that if an identifier is referenced it must have been declared in a textually enclosing block, procedure, cogroup or coroutine or it must have been delivered through a name parameter. In the same way values of variables can be ignored as it is stated that variables may have any value whatsoever.

We may leave out also all information concerning the flow of control inside blocks, procedures and coroutines by stating that they can execute any of their statements independently of the execution so far. But we go still further and leave out the whole program and state that the execution of a statement may be followed by the execution of any statement.

We have left out many details of the execution of programs. It is now impossible to say what is the next action to be done, what are the values of variables, and what identifiers can be referenced. Thus an execution of the model proceeds in a non-deterministic manner. However, given any real execution it is possible to find an equivalent execution of the model. Thus all statements about the model are valid also in real executions.

The model consists of a set of states and rules which define how a new state can be obtained from another. Each state contains all existing procedure, cogroup, and coroutine instances (i.e. executions of a procedure, cogroup or coroutine). Blocks are not included as they can be considered to be a special case of procedures. Two functions, T and D, which will be defined formally later, associate each instance with two other instances. T(x) gives the instance where x is declared and D(x) usually gives the instance which called x. In figures we denote T by double arrows and D by single arrows. Thus single arrows show the dynamic nesting while double arrows give the textual nesting. Each state contains moreover a special instance named P. It represents an underlying operating system and D(P) gives the currently active instance.

Fig. 1 shows a state occurring possibly during an execution of a program containing the cogroup readdata of section 2. This example demonstrates the use of the function D in the case of temporarily suspended coroutines. Instances are labeled with the name of the associated textual unit and they have a subscript to distinguish between different incarnations. The coroutine writechar is active. The coroutine build has called the procedure substructure which has called itself recursively. When writechar will eventually call the coroutine build, build will continue its execution within the second incarnation of substructure.

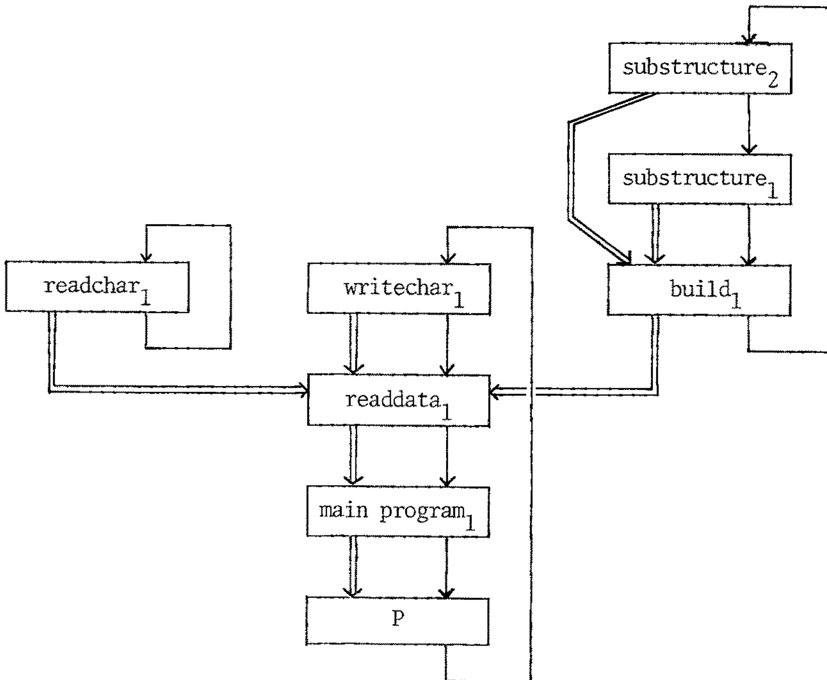

Figure 1: A state of the execution of readdata

We now turn to the formal definition of the model. The set of currently existing instances will be denoted by S. The set S and the functions T and D will be continuously changing during execution and we will give their strict definition later. Here it is sufficient to know that S is a set of instances, P is an instance of S, T is a function defined on S-{P} and D is a function from S into S.

First we define two relations $\Rightarrow$ and $\rightarrow$. Let x and y be two instances. We say that $x \Rightarrow y$ if and only if $x \neq P \wedge T(x) = y$, and that $x \rightarrow y$ if and only if $x \neq P \wedge D(x) = y$.

Let N be a subset of existing cogroup and procedure instances with the intuitive meaning: if an instance belongs to N then it has parameters called by name. The strict definition of N will be given later. Let x and y be two instances. We say that $x \rightsquigarrow y$ if and only if $x \Rightarrow y \;\vee\; x \in N \wedge x \rightarrow y$.

We need also the reflexive and transitive closures of the previous relations. We say that $x \overset{*}{\Rightarrow} y$ ($x \overset{*}{\rightarrow} y$, $x \overset{*}{\rightsquigarrow} y$) for two instances x and y if and only if (1) ((2), (3)) is true.

(1) $\quad x = y \;\vee\; (\exists v)(x \Rightarrow v \wedge v \overset{*}{\Rightarrow} y)$

(2) $\quad x = y \;\vee\; (\exists v)(x \rightarrow v \wedge v \overset{*}{\rightarrow} y)$

(3) $\quad x = y \;\vee\; (\exists v)(x \rightsquigarrow v \wedge v \overset{*}{\rightsquigarrow} y)$

Intuitively, an instance x can contain a reference to an incarnation of an identifier declared in an instance y only if $x \overset{*}{\rightsquigarrow} y$. This rule covers the case of name parameters as well as the case of normal referencing of an identifier in a textually enclosed instance since $x \overset{*}{\Rightarrow} y \supset x \overset{*}{\rightsquigarrow} y$ is a direct consequence of the definitions.

Coroutine instances never belong to the set N even though they may contain name parameters. This is justified by the restrictions concerning actual name parameters of coroutines. They guarantee that an identifier occurring in such parameter can be referenced even directly within the coroutine i.e. when the corresponding formal parameter can be referenced. Thus the definition of $\overset{*}{\rightsquigarrow}$ covers all cases although coroutine instances will never be added to the set N.

The properties we want to prove concern possibilities of referencing and relations between instances. In addition, a state of the model describes only instances and their textual and dynamic nestings. Thus it is sufficient to consider only the effect of activations and deactivations of procedures, cogroups and coroutines.

We have mentioned that the removal of uninteresting details leads to a non-deterministic behaviour. The non-determinism is introduced by choosing non-deterministically one of the operations (5)-(8) below to produce a new state. Of course only such operations can be chosen which are defined on the old state i.e. for which the required instances can be found. Moreover the operations themselves are non-deterministic and

we mark such points with footnotes. Now we give a strict definition of the sets S and N, and of the functions T and D.

In the beginning of execution assertion (4) is true and thereafter the sets S and N and the functions T and D are changed by executing the non-deterministic operations (5)-(8) in a non-deterministic order.

(4) $S = \{P\} \wedge N = \emptyset \wedge D(P) = P$

(5) Activation of a procedure: Let $D(P) = y$ and z be such that $y \overset{*}{\rightarrow} z$ [1]. A new instance x is added to the set S and functions T and D are partially redefined: $D(P) = x$, $D(x) = y$, $T(x) = z$. The instance x may[2] be added also to the set N. The instance x is said to have been activated and it is called a procedure instance. (See Fig. 2.)

(6) Activation of a cogroup: Let $D(P) = y$ and z be such that $y \overset{*}{\rightarrow} z$ [1] and $n \geq 1$ [3]. New instances $g, x_1, \ldots, x_n$ are added to the set S and functions T and D are partially redefined: $D(P) = x_1$, $D(x_1) = g$, $D(g) = y$, and $D(x_i) = x_i$, $2 \leq i \leq n$, $T(g) = z$, and $T(x_i) = g$, $1 \leq i \leq n$. The instance g may[2] be added also to the set N. The instance g is said to have been activated and it is called a cogroup instance. Instances $x_1, \ldots, x_n$ are called coroutine instances. (See Fig. 3.)

(7) Activation of a coroutine: Let $D(P) = y$ and x be a coroutine instance such that $y \overset{*}{\rightarrow} T(x)$ [4]. If there is no unique coroutine instance x' such that x' has been added to the set S by operation (6) at the same time as x and $x' \overset{*}{\rightarrow} P$ then the operation is said to be improper; otherwise if $x = x'$ then the operation has no effect, otherwise function D is partially redefined: $D(P) = D(x)$, $D(x) = D(x')$, $D(x') = y$. The instance x is said to have been activated. (See Fig. 4.)

(8) Deactivation: Let $D(P) = x$, $x \neq P$ [5] and $D(x) = v$. All instances z such that $z \overset{*}{\rightarrow} x$ are removed from the set S and if some of these belong to the set N they are removed also from here. Moreover the function D is redefined to be its restriction on the set S and D(P) is changed to be v. If v is a cogroup instance then this deactivation includes a new deactivation.

According to the above operations it is clear that as long as an instance $x \neq P$ belongs

---

[1] z is chosen non-deterministically.

[2] Whether this is done or not, is determined non-deterministically.

[3] n is chosen non-deterministically.

[4] x is chosen non-deterministically. If such x cannot be found then the operation is undefined.

[5] If such x cannot be found then the operation is undefined.

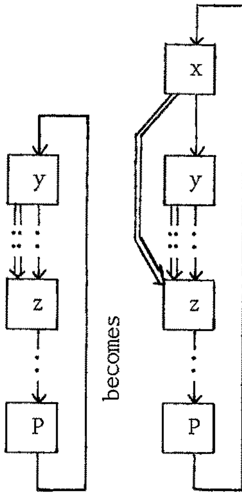

Figure 2: Activation of a procedure

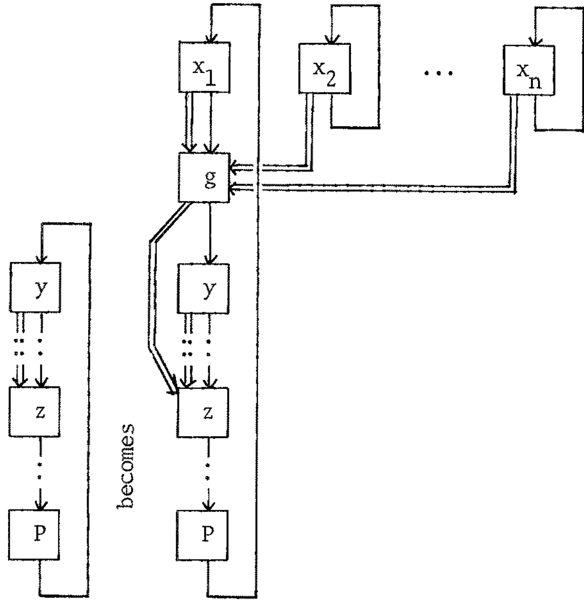

Figure 3: Activation of a cogroup

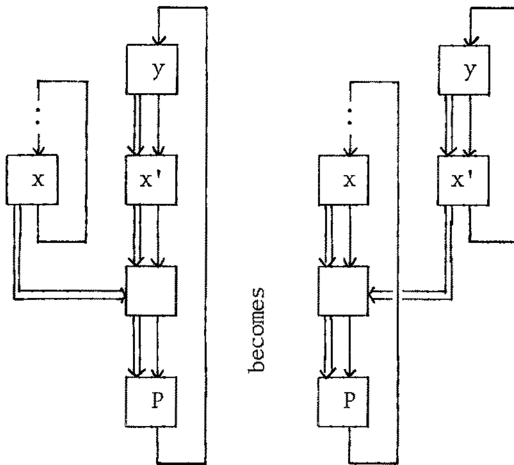

Figure 4: Activation of a coroutine

to the set S the value of T(x) is constant and moreover, if x is not a coroutine instance then also D(x) is constant.

The definition pays no attention to the possible halting of an execution. Especially operations (5) and (6) are defined on every state (by choosing $z = y$). Thus any exe-

cution can extend to an infinite length if no improper activation of a coroutine is encountered. Infinite executions, however, bring no problem as we are interested in properties of states occurring somewhere in an execution but ignore the following states. Moreover, as the above definitions give no rules about halting an execution there is no need to consider halting as a special case.

4. BASIC PROPERTIES

Functions and relations of our model are defined as in the WD-model [7]. The definition of operations (5)-(8) is based on non-determinism but this approach differs from the WD-model only in a philosophical sense. Moreover these operations can be defined with the help of events activation and termination and operations swap and rotate of the WD-model: operation (5) is an activation of the WD-model, operation (6) consists of a series of activations and swaps, operation (7) can be performed by a rotate whenever it has any effect, and finally operation (8) is a termination (or a series of terminations) of the WD-model. Thus we can take the following result from section 5 of [7] as the restrictions made in the proofs are satisfied in our model. These restrictions concern the right to use the operations swap and rotate and are denoted by CS and CR respectively. They are satisfied in our model as $x_i \overset{*}{\rightsquigarrow} T(x_i) \wedge x_i \notin N,\ 2 \leq i \leq n$ in operation (6) and $y \overset{*}{\rightsquigarrow} T(x') = T(x) \wedge x' \notin N \wedge x \notin N$ in (7) if the operation is not improper.

Theorem 1: Unless an improper activation of a coroutine is encountered the following assertions are valid during the whole execution.

(9) $P \in S$

(10) D is a bijection from S to S

(11) $(\forall x)(x \in S \wedge x \neq P \supset T(x) \in S)$

(12) $(\forall x)(x \in S \wedge x \neq P \supset D(x) \overset{*}{\rightsquigarrow} T(x))$

(13) $(\forall x,y)(x \in OC \wedge x \overset{*}{\rightsquigarrow} y \supset x \overset{*}{\rightarrow} y)$

(14) $(\forall x)(x \in S \wedge x \overset{*}{\rightsquigarrow} D(P) \wedge x \neq D(P) \supset D(x) \overset{*}{\rightsquigarrow} D(P) \wedge D(x) \neq D(P))$

where OC is the set $\{x | x \overset{*}{\rightarrow} P\}$.

Assertion (10) means that the function D forms separate cycles on the set S. That particular cycle to which P belongs is called OC (operating chain). Every time an instance is deactivated one (or two) instance belonging to OC is removed from S and some whole cycles are possibly removed. This is a direct consequence of assertion (14) because if an instance $y \neq D(P)$ is removed then we must have $y \overset{*}{\rightsquigarrow} D(P)$ and hence $D(y) \overset{*}{\rightsquigarrow} D(P)$. Thus

also the instance D(y) is removed. Assertion (14) states moreover that $D(y) \neq D(P)$ and so the above proof can be repeated to show that D(D(y)) is removed. Hence it is clear that the whole cycle containing the instance y is removed. However, such instances that contain declarations of variables that can be still referenced are never removed: Let x be D(P) and z be such that $x \overset{*}{\rightsquigarrow} z$. According to (13) we have $x \overset{*}{\Rightarrow} z$ and so $z \in OC$ and thus $z \in S$.

5. RELATIONS BETWEEN INSTANCES

We will now prove that when a coroutine, say a, is activated there is exactly one active coroutine, say b, of the same cogroup. We will moreover find that if control returns to the point where the coroutine a was activated then this return is due to a call for b and not, for example, some containing or contained coroutine. Thus the overall structure of cogroups and coroutines as well as the restrictions imposed guarantee that an activation of a coroutine defines uniquely the coroutine which will be called at the time of return.

The fact that the activation of a coroutine, a in the above example, defines uniquely the coroutine b has important effects on implementations. For example, when a is activated it is known whose parameters will be renewed when control will return: since the return will be due to a call for b, the parameters of b will be renewed. Moreover, as coroutines a and b belong to the same cogroup the balance between co-operating coroutines is maintained i.e. only one coroutine of a cogroup can be active at any time.

To assist proofs we define a set which, as we shall see later, contains all coroutine instances that can be activated and that do not belong to OC. The new set is named C and it is empty at the beginning of an execution. Thereafter the set will be changed by the following additions to operations (5)-(8). The activation of a procedure (5) has no effect on C. The activation of a cogroup (6) adds instances $x_2,\dots,x_n$ to the set C. The activation of a coroutine (7) has no effect on C if $x = x'$, otherwise it adds x' to C and removes x from C. Moreover the activation of a coroutine is said to be improper also if $x \neq x' \wedge x \notin C$. Thus a coroutine can be activated only if it belongs to the set C or if it activates itself. The deactivation (8) removes those instances from C that are removed also from S.

In this section we will follow executions only to a point where an improper activation of a coroutine is met. If such a situation does not occur we follow actions to any desired length.

The following lemma states that exactly one element of the set C belongs to each cycle, excluding OC which contains no element of C. This means that though a cycle (different from OC) may contain many coroutine instances, exactly one of them (the one belonging to C) can be activated, as otherwise the activation would be improper, which we have

prohibited for the present. Moreover it is clear that the coroutine instance which belongs to C will be constant as long as the cycle exists, and so we have an interesting result: at the time of creating a cycle we know what coroutine instance (if any) of this cycle will later be activated.

Lemma 1: Assertion (15) is always true.

$$(15)\quad C \cap OC = \emptyset \wedge \neg(\exists x,y)(x \neq y \wedge x \in C \wedge y \in C \wedge x \overset{*}{\rightarrow} y) \wedge (\forall x)(x \in S \wedge \neg(\exists y)(y \in C \wedge x \overset{*}{\rightarrow} y) \supset x \in OC)$$

Proof: This proof as well as the following ones are based on induction on the number of operations done during execution. We start with the initial state (4) and the condition $C = \emptyset$ given in the beginning of this section and prove that the assertions hold for this state. Then we suppose that we have a state for which the assertions and earlier lemmas hold and prove that the assertions hold for each state produced by any defined operation with any choice of non-deterministic values. Thus the assertions hold for each state in every execution.

In the beginning $C = \emptyset$, $S = \{P\}$ and $D(P) = P$ and thus (15) is true. Now let the state be such that (15) is true. The activation of a procedure has clearly no effect on the assertion because it only adds a new element to OC and does not change the set C. After an activation of a cogroup each new instance x of C will have $D(x) = x$, and as these instances form the new cycles assertion (15) remains valid. If an activation of a coroutine has no effect then (15) is clearly true for the new state. Otherwise the previous "upper" part of OC forms a new cycle which will have one instance in the set C, and the upper part of OC will be replaced by another cycle which, according to the induction hypothesis, has only one instance in C and that instance will be removed from C. So assertion (15) remains valid also in this case. A deactivation may cause troubles only if for some $x \notin OC$ there is y such that $x \overset{*}{\rightarrow} y \wedge y \in C$ and y will be removed. In this case, however, x and y belong to same cycle and according to assertion (14) if y is removed from S then also x is removed from it. Thus (15) is true after each operation if it was true before them. □

Lemma 2 convinces us that cogroups work as we thought: one coroutine instance is immediately above the associated cogroup instance while others belong to the set C waiting for activation.

Lemma 2: Let $g, x_1, \ldots, x_n$ be instances added to S during an activation of a cogroup. Assertion (16) is true as long as g belongs to S.

$$(16)\quad (\exists i)(1 \leq i \leq n \wedge x_i \notin C \wedge D(x_i) = g \wedge (\forall j)(1 \leq j \leq n \wedge j \neq i \wedge x_j \in S \supset x_j \in C))$$

Proof: When the activation in which $g, x_1, \ldots, x_n$ are added to S happens, assertion (16) becomes true and the desired value of i is 1. Next we assume that (16) is true. Activations of procedures and new cogroups clearly have no effect on the assertion.

An activation of some coroutine instance in question retains (16) valid since the searched coroutine instance x' belongs to OC and hence $x' \notin C$ which implies $D(x') = g$. Moreover the activated one must belong to C. An activation of a coroutine other than $x_1, \ldots, x_n$ has no effect on (16).

In the case of deactivation we have three alternatives: the instance $x = D(P)$ does not belong to the set $\{g, x_1, \ldots, x_n\}$, it is g or it is some instance of $\{x_1, \ldots, x_n\}$.

Consider the first case. If g is removed from S then the lemma states nothing any more. If g is not removed from S then the x_i, for which $D(x_i) = g$, is not removed either: otherwise we should have $x_i \overset{*}{\rightsquigarrow} x$ which implies $x_i = x$ or $x_i \Rightarrow g \overset{*}{\rightsquigarrow} x$ as $x_i \notin N$. The former is a contradiction to the hypothesis of this case and the latter is a contradiction to the fact that g is not removed. Thus (16) remains valid.

When assertion (16) is true the instance to be deactivated cannot be g, since then $D(x_i)$ and D(P) would both be g and as $x_i \neq P$ this is a contradiction to the bijectivity of D.

The only instance of the set $\{x_1, \ldots, x_n\}$ that can be deactivated is that x_i for which $D(x_i) = g$, since others belong to the set C and hence cannot belong to OC according to lemma 1. So we have $D(P) = x_i$ and $D(x_i) = g$. The deactivation leads to situation $D(P) = g$ and thus includes a new deactivation. Hence g will be removed from S and the lemma states nothing any more. □

The following two lemmas are needed to get theorem 2, which states that the restriction made in the beginning of this section is unnecessary: no activation of a coroutine can be improper.

Lemma 3: When any coroutine instance x is activated such unique coroutine instance x' which has been added to the set S at the same time as x and which belongs to OC always exists.

Proof: Let $g, x_1, \ldots, x_n$ be as in lemma 2 and $x = x_j$ for some $1 \leq j \leq n$ be a coroutine instance to be activated. Since $D(P) \overset{*}{\rightsquigarrow} T(x) = g$ we have according to assertion (13) that $g \in OC$. The proof of lemma 2 implies that $D(P) \neq g$ and hence according to lemma 2 we can find i such that $D(x_i) = g$ and therefore $x_i \in OC$. Since $(\forall k)(1 \leq k \leq n \wedge k \neq i \wedge x_k \in S \supset x_k \in C)$ and $C \cap OC = \emptyset$ we have $(\forall k)(1 \leq k \leq n \wedge k \neq i \wedge x_k \in S \supset x_k \notin OC)$. Now we can choose $x' = x_i$ and this is the only one satisfying the requirements of the lemma. □

Lemma 4: When any coroutine instance x is activated then $x \in OC \cup C$.

Proof: In order to be able to activate x we must have $g \in OC$, where $g = T(x)$. According to lemma 2 either $x \in C$ or $D(x) = g$ (since $x \in S$ as the activation of x is defined). In the latter case we have $x \in OC$ and so $x \in OC \cup C$. □

Theorem 2 is a direct consequence of lemmas 3 and 4.

Theorem 2: An activation of a coroutine is never improper (as defined in the operation (7) and supplemented at the beginning of this section).

According to the discussion after theorem 1 each cogroup instance belongs to the set S as long as its coroutines can be referenced. We must, however, still prove that these coroutine instances do exist i.e. also they belong to S. This is done in theorem 3.

Theorem 3: Let $g, x_1, \ldots, x_n$ be as in lemma 2. Assertion (17) is true as long as g belongs to S.

(17) $(\forall i)(1 \leq i \leq n \supset x_i \in S)$

Proof: When the cogroup is activated assertion (17) becomes valid and it can become false only through a deactivation. Let $D(P) = y$ be to be deactivated so that x_i will be removed from S for some $1 \leq i \leq n$. We claim that g will also be removed from S. According to the definition of deactivation we have $x_i \overset{*}{\rightarrow} y$ and thus either $x_i = y$ or an instance v exists so that $x_i \Rightarrow v \overset{*}{\rightarrow} y$. This follows from the fact that $x_i \notin N$ as it is a coroutine instance. Since $T(x_i) = g$ we have in the latter case $v = g$ and then $g \overset{*}{\rightarrow} y$ which implies that also g will be removed from S. In the first case $x_i = y$ we have $x_i \in OC$ and so $x_i \notin C$ and hence $D(x_i) = g$ which implies that the deactivation of x_i includes the deactivation of g which therefore will be removed from S. □

As we have already noticed theorem 1 implies that each time an identifier is referenced the instance containing the declaration of that identifier belongs to OC. Moreover each label can be thought of as being declared in the innermost block in which it occurs. Thus at the time of a reference to a label the instance containing the declaration of that label and the corresponding program point can be found in OC. This means that the effect of a go to-statement can be modeled by zero or more deactivations and hence our former results remain valid.

To complete the proofs we show that the dynamic ancestors of an instance remain constant.

Lemma 5: Let $x \neq P$ be an instance. Then we have a unique y such that $x \in OC \supset x \rightarrow y$.

Proof: If x is a procedure or cogroup instance the value D(x) is constant all the time x belongs to S. If x is a coroutine instance D(x) can be changed but according to lemmas 1 and 2 $x \in OC \supset x \rightarrow g$ where g is the corresponding cogroup instance. □

Theorem 4 follows easily from lemma 5.

Theorem 4: Let x be an instance. Then we have unique $y_1,\ldots,y_k$, $k \geq 1$, such that $y_1 = x$, $y_k = P$ and $x \in OC \supset (\forall i)(1 \leq i < k \supset y_i \rightarrow y_{i+1})$.

6. CONCLUSION

We have sketched a method and notation for expressing coroutine control and given a model for its executions. Using this model we have proved that the balance between co-operating coroutines is maintained in every execution and that it is impossible to reference terminated coroutines and variables declared in any terminated instance.These results convince us that the use of the coroutine system is safe: invalid references cannot occur and the nesting of coroutines cannot lead to unsuspected situations.

The result concerning the balance between coroutines of any cogroup obtains more importance if the coroutine system is extended to include some advanced concepts. For example, typed coroutines and "subcoroutines", which always return control to their callers, are easier to define and use when co-operating coroutines cannot call each other in unobvious ways [6].

The restriction imposed on actual name parameters transmitted to coroutines has dual effect. On one hand it guarantees the balance between coroutines and the safety of their use, on the other hand it prevents the referencing of identifiers declared in terminated instances. If the coroutine system is incorporated into a language using retention instead of for example stack techniques in memory management then the second effect is not needed and can be removed by mitigating the name parameter restriction not to cover arithmetic and Boolean variables other than formal parameters.

REFERENCES

1 Burroughs B6700/B7700 Extended Algol Language Information Manual. Burroughs Corporation, Form 5000128, 1972.

2 Conway M. E.: Design of a Separable Transition-Diagram Compiler. Comm. ACM 6,7 (1963), 396-408.

3 Dahl O-J., Myhrhaug B., Nygaard K.: SIMULA 67 Common Base Language. Norwegian Computer Center, Publ. No. S-2, 1968.

4 Gentleman W. M.: A Portable Coroutine System. Information Processing 71 (ed. C. V. Freiman), North-Holland, 1972, 419-424.

5 Krieg B.: A Class of Recursive Coroutines. Information Processing 74 (ed. J. L. Rosenfeld), North-Holland, 1974, 408-412.

6 Sajaniemi J.: A View of Coroutines. In preparation.

7 Wang A., Dahl O-J.: Coroutine Sequencing in a Block Structured Environment. BIT 11,4 (1971), 425-449.

8 Wulf W. A. & al.: BLISS Reference Manual. Carnegie-Mellon University, 1970.