THE AUSTRALIAN NATIONAL UNIVERSITY

DEPARTMENT OF COMPUTER SCIENCE

THE RUN-TIME STRUCTURE OF SIMULA 67

by

C.W. Johnson

TR-CS-77-01

The Run-Time Structure of Simula 67

C.W. Johnson
27.2.77

Computer Science Department
Australian National University

The control structures of Simula 67 are described in terms of their effects on a branching stack of class, procedure and block activation records. The description is intended as an aid to understanding the capabilities of Simula 67 and the structure of a Simula 67 virtual machine implementation. Storage allocation for this implementation is also discussed.

§1. Simula 67 Semantic Possibilities

Simula 67 is a general purpose and simulation language, based on Algol 60. The simulation capabilities are defined in terms of the general language features, so only the general purpose features will be described here.

Simula contains nearly all of Algol as a subset*. Added to Algol is a new kind of program component, the class, somewhat resembling both the procedure and the record structure of Pascal and Hoare. Multiple instances of any class may exist at any time. Unlike a procedure, which can also have multiple instances in existence, the instances of a class can "know about" and communicate with each other and instances of other classes, and control can pass between them as coroutines. Parameters may be provided in generating a class instance: these parameters and variables declared in the class body (known collectively as "attributes" of the class instance) are read- and write-accessible from outside the class instance. Different instances of a class are different entities and have their own copies of the attributes of the class. (As different instances of the one procedure have their own parameters and local variables.) The coroutining ability means that any class instance can suspend its own execution at any stage, some other class instance carrying on from the point at which it was last suspended itself. (See Wegner §4.10.3 for a general discussion on coroutines.)

---

* Unless otherwise made explicit, Simula 67 will be referred to as Simula, Algol 60 as Algol.

A new type of variable is also introduced to allow reference to be made to instances of a class. A direct pointer to an instance, with more restrictions on use than the Algol 68 "ref amode", is used. ref is used as the corresponding mode constructor (cf array) and any ref declaration is qualified with the name of some class. A ref variable with a particular qualification cannot refer to an instance of any other class, which constitutes a very powerful security feature.

e.g. ref (point) p1, p2;

declares p1 and p2 to be ref variables that may refer only to instances of the class point. The operations of accessing and changing the attributes of a class instance and passing control to it are expressed using references and reference expressions yielding references. A dot notation accesses attributes - it may be read as 's. E.g. FRED.AGE is the attribute AGE of the class instance currently referenced by FRED, or FRED's AGE.

Control is passed by "resume (class instance reference)".

```
e.g.  a:= p1.firstpara;
      p2.second:= a+b;
      p2:-p1;
      resume (p2);
```

("Denotes" (:-) is used to assign references to reference variables.)

An instance of a class is generated by means of the construct new; "new classid (actual parameters)" is a reference expression yielding a reference to a new and unique instance of the class "classid". It would normally be used as the right hand side of a denotation statement:

```
e.g. begin
     class A (x,y); integer x,y;
           begin
           comment body of A;
           end;
     ref (A) aninstance;
     aninstance :- new A(3,17*27);
      :
      :
```

which leaves aninstance denoting the new instance of class A, having the values 3 and 17*27 for its integer attributes x and y.

When a class instance is generated it is in an "attached" state, and can be regarded as similar to an untyped procedure in its behaviour. It is not yet a detached object in its own right, and should it execute a "resume" this is regarded as originating from within the generating class instance or prefixed block, to which the new class instance is attached. It becomes a detached object when it first executes a "detach" (formally, a procedure with no parameters). Control then returns to the object generator (i.e. part of the function of "detach" is as an explicit return) and the new class instance now has existence independent of the block that generated it.

§1.1 Control structure - general

A group of class instances can operate as coroutines using the "resume" construct. Resume is syntactically a typeless procedure with a single parameter that is of "type" ref (anything). This parameter must refer to some detached class instance. The execution of a resume statement will cause the position of the next statement to be saved as resumption point for whatever object (main program or class instance) is active, and control to pass to the previously similarly saved resumption point of the object referenced by the parameter.

e.g.
```
begin ref (A) B;
class A
    begin ...
    detach;
    outtext ("bah"): outimage;
    end;
  .
  .
  .
comment generate an instance of A, by;
  B:- new A;
      .
      .
      .
  resume (B);
      .
      .
      .
```

The resume will pass control to the statement following the detach (which was the point at which control left previously) in the instance of A referred to by B and "bah" will be printed. The object that is active at the time of the resume, in this case, is the main program, which behaves like a detached object in many respects.

If two class instances (of the same or different classes) each contain resume statements with a reference to the other as parameter, then full coroutining can occur:

```
begin
ref (C) consumer; ref (P) producer;

class C;
     begin
     detach    ; comment do nothing when created;
     while true do
          begin
          consume buffer;
          resume (producer)
          end
     end;

class P;
     begin
     detach;
     while moreinput do
          begin
          fill buffer;
          resume (consumer)
          end
     end;

producer:- new P;  consumer:- new C;
resume (producer);
terminal processing
end;
```

This will have the effect of generating one instance of each class, and then starting up loops that will pass control back and forth between the two instances until "moreinput" becomes false. Each time control returns to the instance of C referred to by consumer, it takes up where it last left off. The same would be true in a program in which many objects had control in the meantime.

What will happen here when "moreinput" is false? Control will then reach the terminal end of the only instance of P we have created. In a normal Algol procedure this would signify that control passes back to the point of call: but returning to the initial generating expression (for the second time - return has been made there previously, on executing "detach" during generation), or the most recent object (consumer) that resumed this P, will not allow us ever to get out of our system of coroutines (except with great difficulty and unobvious global variable setting and testing). Instead, a return is made to the main program - in particular, to the resumption point that was saved for the main program when it did "resume (producer)" to start the coroutines operating. An exactly similar return to the main program can be had in another way; by an already detached class instance executing another "detach" statement. Such a "detach" saves the resumption point and the class instance can later be resumed at that point.* Once its terminal end has been passed, however, a class instance has no resumption point and may not be resumed. Its attributes remain accessible.

When one class declaration is embedded in another then we might expect either of two things to happen with instances of the two classes. Either there is a single "main program" (the main program) to which a detach from an already detached object will return control: or "main" programs, like many other Algol/Simula concepts, can be nested; and a "detach" will return control to the most immediately enclosing "main" program. Simula provides for such nesting to take place, and the "most

---

* The two uses of "detach", from an attached and from an already detached class instance, are entirely separate operations and only have in common the saving of the resumption point for the class instance.

immediately enclosing" main program is decided from the form of the program text. A construct known as a prefixed block (which is an Algol-like block whose begin is preceded by some class identifier) allows the choice of positioning of "main programs" in the program structure - each prefixed block is a "main program". An easier view may be that a "detach" from an already detached class instance will return control to the smallest enclosing prefixed block, and the main program is a prefixed block with an invisible prefix.*

Another part of the meaning attached to resumption points complicates the matter further, and also provides extra motivation for the prefixed block concept. With no prefixed blocks appearing in a program, all class instances in that program are, for most purposes, operating on a single level. They are all coroutines to each other: each has its own, distinct, resumption point, and a detach from any one of them will return control to the same point in the main program. The nesting of class declarations is not sufficient for the nesting of systems of coroutines unless prefixed blocks are used. There will be access (visibility) differences between the objects, because of scoping rules, but otherwise no differences in their control environments.

We have here a flat picture of control transfers - as if we were working in the (control only) equivalent of a main program and subroutines in FORTRAN. Each detached object, like a subroutine, can contain no control detail, and cannot, for instance, consist of an inner system of coroutines. If prefixed blocks are included, this picture acquires depth, and we gain the ability to nest systems of

---

* Note that the operation of "detach" in Dahl and Hoare is not the same as in Simula 67.

coroutines within the objects that constitute components of coroutine systems themselves - as would be expected in the spirit of Algol.

The additional concept that causes such a system to behave as a complete unit within another system is that each system unit is perceived, from outside that unit, as containing a single resumption point. The resumption point of such a unit system is that of the most recently active object that caused control to leave this unit system by resuming some object outside it. A class instance can only contain a sub-system of coroutines by embedding their class declaration(s) in a prefixed block within its body. The scoping rules ensure that the resumption point of the instance must be within this prefixed block for any instances of the embedded classes to exist. On resumption of this class instance the most recently active part of the system consisting of this prefixed block and its enclosed class instances regains control. This most recently active part must be that which caused control to leave this same system previously. Consequently, an attempt to resume the embedding class instance from within one of the embedded instances will behave as a null (skip) operation: it resumes the whole sub-system at its most recently active point, which is just that "resume" being executed. To resume execution of the embedding class instance a "detach" from within the sub-system will suffice: the prefixed block to which this returns control is also the resumption point of that class instance.

§1.2 Control Structure - details

Straight and Branching Stacks

The operation of a group of detached class instances and prefixed blocks can be described as follows, as a variation on the stack of activation records of Algol.

Instead of a single stack of activation records Simula needs a stack that can sprout branch stacks, from points in the stack below its current top. Each branch can grow and branch in the same way as the main stack. Only one branch of the whole tree can be operating at any time, the "active" activation record being on the tip of this branch, and control may pass among the branches without destroying them.

When a new object is created, it behaves, for control purposes, as if it were attached to the currently active branch of the stack. ("Attached" here has the obvious meaning which coincides with my previous use.) For data access purposes it will behave as if it belongs in the tree on a new branch that originates at the level in the stack where its class declaration occurs. Once the object executes a "detach" it is pulled back to this point for control purposes as well.

Compare this with an Algol procedure activation. When an instance of the procedure is generated (by calling the procedure) an activation record is put on top of the active branch (there is only one) of the stack. For data access, however, the procedure must be thought of as an appendage to this straight stack at the level the procedure was declared. Because the procedure activation disappears once control returns to the generating block, there is no need to grow an explicit branch from lower down in the stack - space allocated at the top of the stack for the procedure activation record may be recovered before the generating block has opportunity to create any further block instances. The conceptually branching tree of data access paths

in Algol can therefore be realized within a strict stack discipline: Simula must have something more than a stack to allow the preservation and resumption of exited class instances.

The branching stack reflects the textual organization of the Simula program, nodes of the tree being activation records of block instances containing class declarations. The semantics of the sequencing control transfer statements can be demonstrated on this branching stack.

A simple tree model

As a simplification, let us assume that the branching points, or nodes, of the branching stack tree can be only at prefixed block activation records (including the main program block). This implies that class declarations can only appear immediately in such blocks. At any node there will be a number of branching substacks. One substack will be the continuing "main program", containing the activation records for the prefixed block and its sub-blocks: the others will each be an instance of a class declared within the prefixed block and its sub-blocks. Procedure activations and attached class instances form part of the sub-stack to which they are attached.

Just as the operating block instance will be at the top of some sub-branch of the tree, so the resumption point of any branch stack (for each branch is a detached object, and has a resumption point) must be at the top of its sub-stack (a branch cannot grow while control is not in it). The main program branch at any node also has a resumption point, reached by a "detach" from one of the other branches at that node as discussed at the end of §1.1.

Control transfers around the tree are limited by the scoping rules. Each branch (except a "main program" branch) must have a reference that

identifies it: because the class declaration for this branch appears in the prefixed block that is the node, the only references to it must be within the branches (and their sub-branches) at this node. Hence a "resume" can only reference a class instance that emanates directly from some node that supports (i.e. has a chain of branch, sub-branches etc. leading upwards to) the current operating point.

If a branch does have a number of sub-branches growing from it, then one of these will contain the "life" of the branch, the others being "dormant" as far as the parent branch is concerned. The resumption point of the branch is that of this live sub-branch - which may be either a "main program" or class instance sub-branch. This live sub-branch is the most recently active sub-branch from the node.

(To re-iterate: a branch is a class instance or the main program, and its associated normal blocks, procedure activations called from it, and still attached class instances; branches come from nodes, which are instances of prefixed blocks or the main program. Detached class instances with declarations appearing in a particular prefixed block are branches from the node that is that prefixed block's activation record. The prefixed block nodes occur within the stack branch of class instances that textually enclose them. There is one privileged branch at any node that is the continuation of the prefixed block (as if no class declarations had appeared to make the stack branch many times at that point) which can have no reference made to it; all other branches, i.e. all class instances, must be referenced to them from somewhere. A reference to some class instance includes the whole system of detached class instances whose declarations it contains, and their contained class instances, and so on. The complete subtree system is an "object".)

Transfers of Control

If control is passed to some point in this tree (by a "resume (reference to a branch/class instance)") then that branch is followed upwards. If a node is encountered, then the "live" sub-branch is followed similarly. When a tip of the tree is reached (i.e. the top of some sub-stack) the activation record at its tip contains the resumption point for the object referenced by the resume statement. It may be a class instance branch or a "main program" branch, and many nodes may be traversed along either kind of branch before the tip is reached.

All the part of the tree that supports this newly reached tip may be regarded as "active". It is composed of instances of blocks, prefixed blocks and classes that are global to the newly operating block instance and hence can be thought of as containing the operating point. This should be amended to "dynamically containing", perhaps, since procedure calls and attached class instances may form part of this chain out of their purely textual positions. Should a transfer of control take place by the operating object's resuming some other object, or executing a detach, then some new line of activation records from the base of the tree to another tip will start operating in the same way. The old operating chain branches will remain the "live" branches from their nodes, except the node that supports both the old and new operating lines. At this node the newly active branch holds the life of the node's supporting branch, because it is more recently active than the branch system that just relinquished control.

That there must be some such shared node in the old and new operating paths of activation records from base to a tip is obvious - for all branches ultimately must spring from the lowest node in the tree. But also because of the scoping rules governing visibility of class declarations the active instance is unable to "look inside" non-active branches of the

tree, and can only "see" its own, operating, nodes. The only branches that can be named, by the scoping rules, are those that come out of an operating node - and this node is the one that is shared by the old and new operating chains. The "main program" branch that can be "detached" to from a detached class instance branch shares a common node with that branch - the prefixed block that is the base of the "main program" branch, containing the declaration of the class instance.

Growth and decay of the tree stack

Each branch of the stack will grow and shrink much like an Algol stack. New branches at nodes lower in the tree than the operating tip-stack can be created, however: and the effect of shrinking a branching stack must be determined. Once the shrinkage of a tip-stack uncovers a node, then the action depends on the type of branch that is operating. If it is the main program sub-stack, then uncovering the node means we are leaving the prefixed block (at the node) that contains the declarations of the classes whose instances are the other branches. The scoping rules ensure that once we leave this declaring block, no references to these class instances can exist. Without any reference to it a class instance can disappear, and so as we delete the activation record at the branching point, we can delete all the branches as well. No branch is left in mid-air with no support.

If we were on a class instance sub-branch then the effect is quite different. Passing through the terminal end of a class instance, which is what we are doing to uncover the activation record at the supporting node, does not mean the class instance ceases to exist, only that it is no longer active or capable of being reactivated. Reaching the end is treated as an implicit "detach" - control passes to the resumption point in the main program branch, its "live" sub-branches being followed to a tip that can operate.

Unrestricted placing of class declarations

This branching stack explanation was based on the assumption that the branching nodes could only be at the activation records of prefixed blocks. However, class declarations can appear in blocks of any kind, and some refinement is necessary to explain the action of a program without this restriction. Statically, for identifier access, the stack-tree can be pictured in exactly the same way, with branching nodes appearing at the declaration points, which are now any block's activation record. The dynamic stack tree looks slightly different, and control transfers will behave as if the tree branched only at the prefixed block points nearest below the points of actual declaration in the static tree. Thus we have a picture of two trees overlaid on the same block activation records, each of slightly different branching patterns (but not radically enough to cause any reversal of the "supports" relationship between two blocks in the tree). A highly confusing picture, indeed. This separation of the static and dynamic trees makes a general Simula program very hard to describe, and my examples will be limited to a single level of class declarations, immediately within each prefixed block. For these examples the two tree structures coincide exactly. More general declaration structures are left as an exercis for eager readers only.

Comparison with Simula Common Base Terminology

The Simula Common Base Language definition uses "quasi-parallel system" to refer to what has been mentioned here as "subtree", and uses the concept of quasi-parallel systems being contained in other quasi-parallel systems, as we have subtrees as branches of lower level subtrees. The "main program" of a quasi-parallel system is our "main program sub-branch". Attached and detached class instances have the same meanings.

§1.3 Program example - with no prefixed blocks

```
begin
Boolean procedure finished;
     begin comment what you will; ... end;

procedure move (wnotb); Boolean wnotb;
     begin comment generate and perform a move for "if wnotb then white
               else black"; ......
     end;

class player (colour); Boolean colour;
     begin ref (player) opponent;
     detach ; comment do nothing when generated;
     while not finished do
          begin integer pdummy;
          move (colour);
          resume (opponent)
          end;
     outtext (if opponent.colour then "whitewon" else "black won");
     detach; comment stop the game;
end of a player;

ref (player) black, white;
while moregamestoplay do
     begin integer dummy;
     black :- new player (false); white :- new player (true);
     black.opponent :- white; white.opponent :- black;
   comment set up initial positions on board; .....
     resume (white); comment start the game;
     display final position; comment come back here at end of game;
     end
end
```

This program establishes a quasi-parallel system (group of sub-branches) of two instances of "player", with the main program acting as supporting branch of the system. The two instances of the class player are referenced by "black" and "white" in the main program, and as "opponent" in the other. When one player has "moved", using some appropriate global procedure, it resumes the operation of the other player - and so on until the game is "finished". The initial "detach" within the body of player detaches the object from its generating block instance (in this case, the main program while block) and its dynamic and static tree positions become the same (in this case, a branch at the main program level), control returning to the generating block instance. Resuming "white" then passes control back to that referenced instance of player. The main program sub-branch of the only node we have here includes

two activation records - that of the main program, which is at the node, and its contained "main program" continuation, which is the while block. At present, the other two branches are each a stack of one activation record: when "white" is resumed, control enters its inner while block and its branch grows to a stack of two. Once "black" is resumed (as white's opponent) it does likewise and these two branches continue to have two activation records in each (except when "finished" is active, when there are three)(fig. 1 shows the activation records at this point) until external reference to them is lost (when "white" and "black" refer to new players) and they cease to exist. The main program sub-branch is returned to only when one player detects that the game is "finished" - and hence (we assume) its "opponent" has won. A further "detach" effects the transfer, and control leaves the quasi-parallel system of the two players. The variables of the two players are still accessible via dot-notation, and the quasi-parallel system is still restartable, at this stage. We could add more code to detect an illegal position, for instance, and leave the code to correct such a position in the main program, then resuming the quasi-parallel players' system with the corrected position.

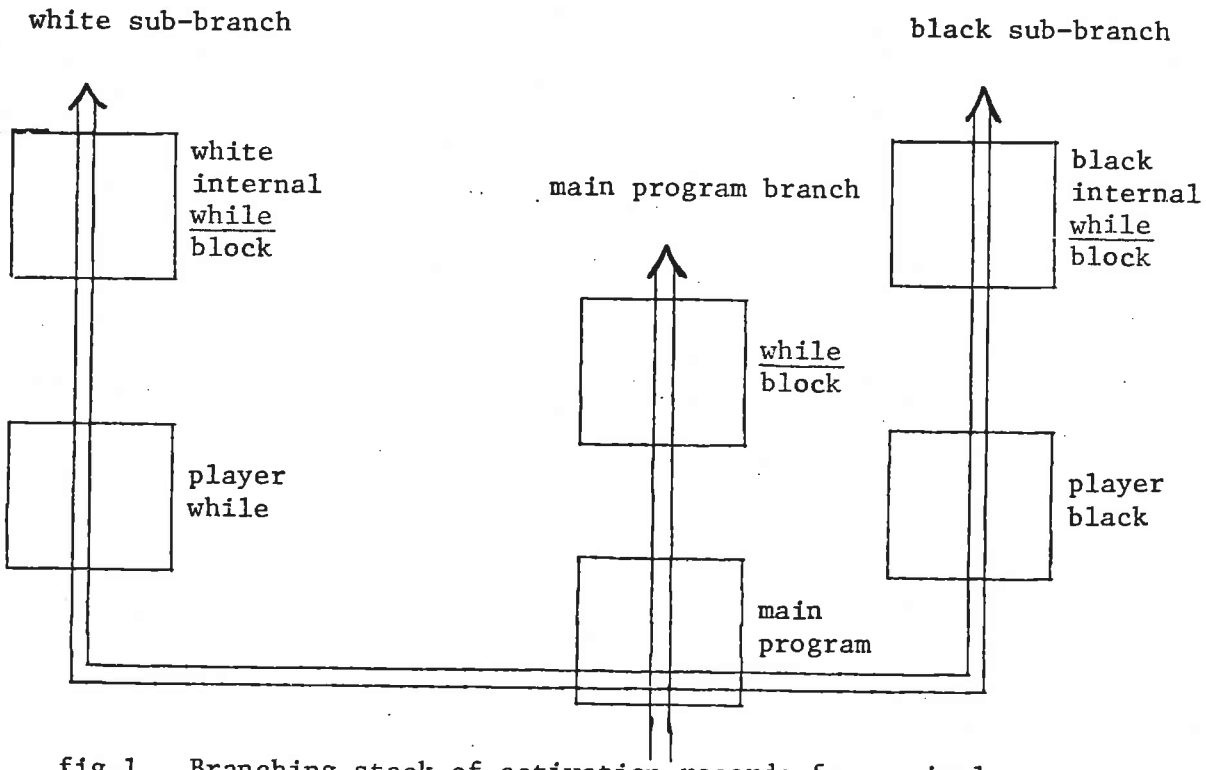


fig.1. Branching stack of activation records for a simple coroutine program.

§2. Terminology

Many of the terms used in §1 are standard Simula usage - but, since some are not, the language of the Simula Common Base Language is covered here. Some additional semantics and the effects of certain constructions are also mentioned.

An instance of a class, while under the influence of the generator new, is said to be "attached", and remains so until it executes a "detach". Until this time it behaves exactly like a pure procedure (with parameters called by value). A further detach will have the result of passing control to th "main program" of this "quasi-parallel system", while the "object" remains "detached", as it was before. Once control passes through the final end of an object, or leaves via a go to, that object is "terminated" and a resume with it as destination is illegal: control can never return to it. Passing through the final end has the effect of an implicit detach. Thus a class declaration with no statements in its body takes its parameter values on generation and then immediately becomes "detached" and "terminated", its "attributes" (formal parameters and outermost body-block-declared variables) accessible but resumption of control not possible. This use of a class is very similar to a pure data structure definition - a record (Hoare) or struct (Algol 68).

An inactive but non-terminated class instance is "suspended".

A "quasi-parallel system" consists of a number of detached class instances that are textually contained ("enclosed" - but this term is not used consistently) in a single "prefixed block". This prefixed block is the node, "main program", of the quasi-parallel system. A prefixed block is an Algol block, except that it is "prefixed" by the name of some class (e.g. player begin ... end). It behaves as a non-referenceable detached object, in that it may have a resumption point of its own, and has the attributes of

the class named in the prefix available within it. (Normally the useful attributes are in the form of procedures and class declarations.) A class declaration may also be prefixed by the name of some other class: the attributes of the prefix class become concatenated with those declared for the new class, enabling the creation of hierarchical data definitions. Reference variables "qualified" by the prefix class may also refer to instances of the prefixed class. This concept is carefully hedged with a number of restrictions designed to remove ambiguity and simplify the run-time structure.

Because these prefixing features are dealt with mainly at compile time, they will not be considered here, except briefly in §5.

§3. Run-time System

§3.1 Description

Simula does not permit the strict stack-based run-time structure that can be used for Algol. In Algol, block instances are created and deleted by a strict "last-created is first-deleted" rule. Scopes of identifiers are strictly nested, and no identifier or block instance coming into existence because of the dynamic entry of some scope has any meaning or existence once that scope has been left. This allows a stack of activation records to be used.

Simula however allows the creation of block instances which "belong" further down the stack and may continue to exist when the current scope has been left. A stack discipline is therefore not possible for Simula. A branching stack must be used, which cannot be directly represented in linear storage. Explicit pointers for down-stack identifier references and up-stack "live" control references must be used. Tracing chains of pointers between activation records will be the equivalent of traversing the tree of branching stacks.

§3.2 Components

The implementation described by Simula's designers (in Simula Implementation Guide) is based on a driver technique. A single copy of the code for each block exists - the "prototype". A block instance is represented by two parts, containing a reference to the prototype: a driver, known as a "notice", and a data area, known (somewhat confusingly) as an "object". The notice contains all the information needed by the run-time system for inter-block control flow and data access - i.e. a description of both static and dynamic environment of the block instance - and an identification of the kind of block (detached, prefixed, etc.) the notice corresponds to. None of this information is necessary once a block instance terminates, as no further execution of code in this activation of the block can be performed, and hence no outgoing control or data access from it.

The attributes of the instance are still accessible, via dot notation, but these are stored in a separate data area, the object. All notices are the same size = any block instance must have linkage information and identification: objects vary in size, depending on the number and type of declarations in the block. A terminated instance will in general possess no notice, while it may possess an object. If the two types of items are kept in separate areas, a simpler storage allocation scheme can be used for the higher turnover equi-sized notices, and a more complex fragment collecting scheme used for the varied-size objects, less frequently. The driver is sufficient to show the control aspects and later diagrams will show only drivers in the branching stack.

In Algol 60 the information in an activation record that is needed to allow access to non-local variables and actual parameters, and to transfer control to and from procedures and enclosing blocks may be expressed

in two pointers to other activation records. A static link is necessary to reflect the textual structure of the program, allowing non-local variables to be accessed: and a dynamic link to reflect the structuring of the program in operation, allowing parameter access and control transfers.

The static chain may be implemented as an inverted index to the activation records (a "display") - this may also be done in Simula 67. Most of the discussion in the Implementation Guide is based on an actual chain, though many of the programs there concentrate on a display index - the two representations are equivalent, and chain concepts will be shown here.

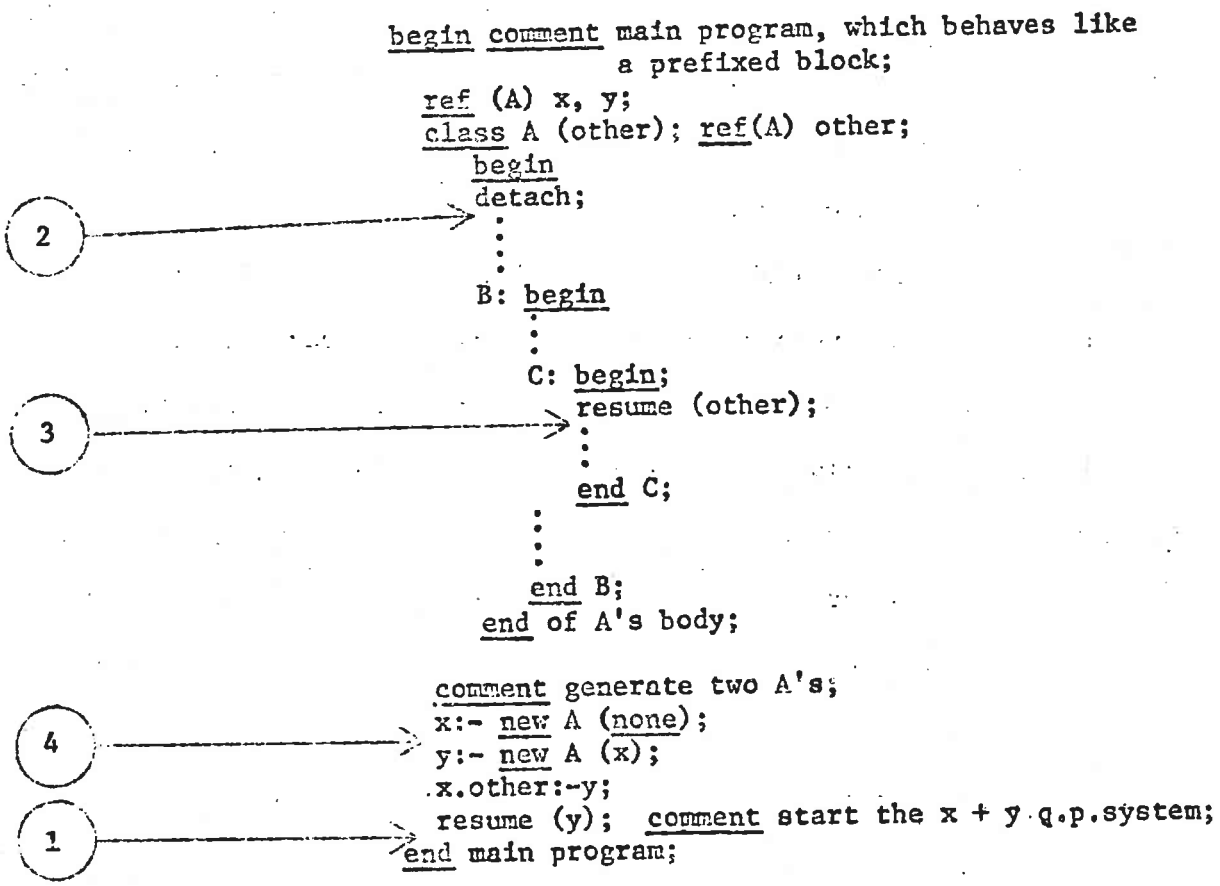
§3.3 Implementation

The driver for a block instance of any kind in Simula contains the static and dynamic links as for Algol. A detached object needs no dynamic "return" link, however, once it has become detached from its creating block instance. All parameters of a class instance are passed by value or reference, which removes the other reason for a dynamic pointer into the creating environment. The dynamic link of a detached class instance is therefore able to be used for a conceptually distinct purpose, never needed in an attached instance: it points up-tree, indicating the live sub-branch forming part of a chain to the live tip activation record. The dynamic link of an active detached block instance is effectively none - any live sub-branch above it must also be active, and there is no need for the up-tree link. That of a suspended class instance indicates the base of some higher sub-tree (enclosed class instance) - which is live - or some block instance at a tip of the tree. This block instance contains the resumption point of the suspended object. Sub-blocks and other attached block instances have normal Algol-like dynamic links to their creating block instance.

Each driver also contains a restart point, and identification of kind of block. A sub-block's driver does not ever use its restart point - the resumption point for its parent detached block instance is held in that block instance's driver. (This means that the restart point stored in a driver need not necessarily indicate a point in this block's code prototype.)

The program example following (Partprogram) illustrates some simple run-time structures. Not all "objects" (corresponding to data areas) are shown - inessential detail in the drivers and their links is suppressed. The program declares two ref(A) variables and a class A with a single parameter. Note that labels in this program serve to identify the blocks that follow, and are not used for transfer of control.

example: Partprogram



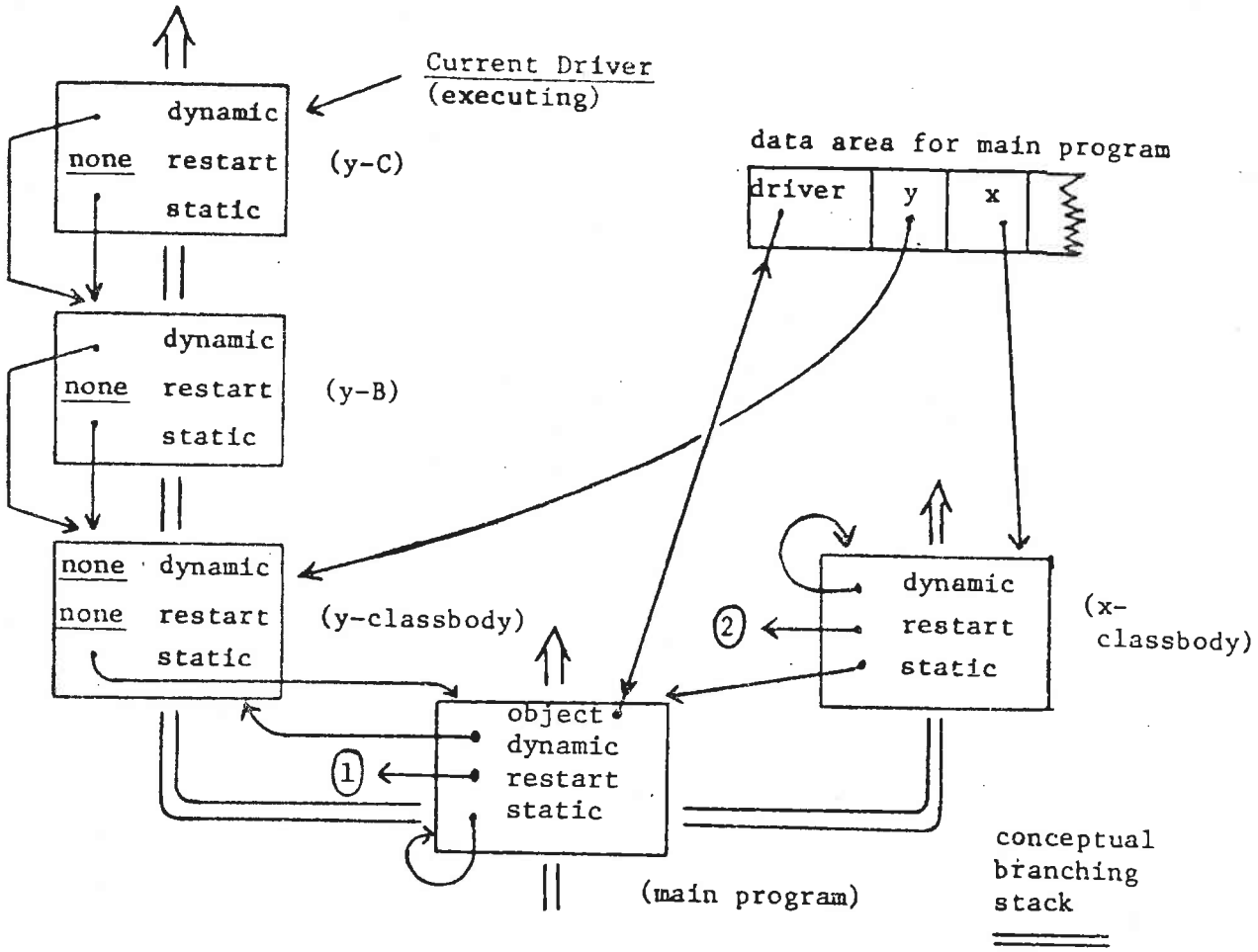


fig. 2. Before "resume (other);" in y.

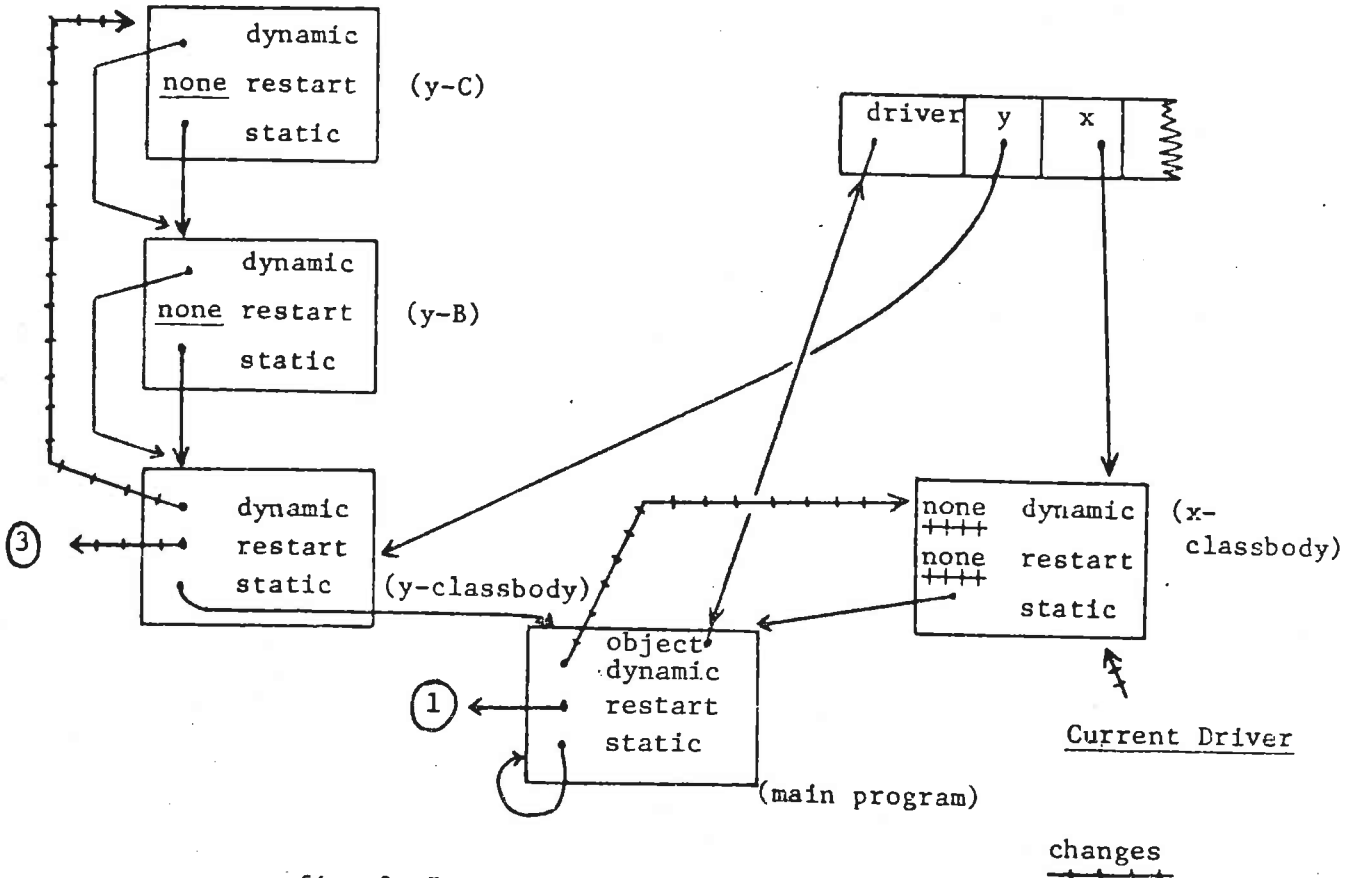


fig. 3. Immediately after x has resumed.

The body of the declaration of A has an immediate detach, and two nested sub-blocks B and C. The innermost, C, has a resume with the class parameter as destination. The main program generates two instances of A, which are referred to by x and y. Each instance of A executes its detach, i.e. suspends itself, becomes detached, and returns to the generating statement. The two instances are set up here so that "other" of each refers to the other. none is a reference constant, that may be assigned to any reference variable (regardless of its qualification). The instance of A referred to by y (or, loosely, "y") is resumed by the main program. It restarts after the detach statement, enters sub-block B - creating an instance of that sub-block - then sub-block C, and executes "resume (other)" - equivalent to "resume (x)". The situation is shown just before that "resume" is encountered (fig. 2), and again after x has resumed execution (fig. 3).

In this simple case the main program block fulfils several duties because there is no extended hierarchy of quasi-parallel systems in which the dynamic pointer of the main program block would not necessarily be affected by such a "resume", as we have here. In all cases, however, the pointers in the class bodies' drivers and sub-blocks' drivers have some general properties. For a class body driver, while attached, the dynamic pointer is to the driver of the generating block instance, and its restart point is to the generating statement in that instance. When detached, the dynamic pointer is always to the innermost block instance of this class instance - the live sub-branch or top of a tip-stack containing the restart point. The restart point of any sub-block is always none, and of a procedure body it is the calling statement in the calling block instance, which its dynamic link will indicate.

§3.4 Maintenance of the implementation structure

The dynamic link of a detached object is used to indicate the dynamically innermost block instance of that object, i.e. the block instance, containing the object's resumption point, that will become the Current Driver if a resume is effected with this object as destination. The block instance to be reactivated lies at the tip of some branch of the stack-tree, reached by a chain of dynamic links passing through each detached object supporting it. This chain need only pass through class instance and some prefixed blocks, avoiding sub-blocks and indicating only the base of each branch that is live. The last link in the chain indicates the block instance to be the new Current Driver - which can be a sub-block.

This chain is maintained in the following way by new, detach and resume.

new:

The newly generated class instance is connected as if it were a procedure activation. The dynamic link is to the driver of the block instance containing the new: the restart point is the succeeding statement in that block's code. The currently active driver is that of the new class instance.

detach from an attached class instance:

Control will return to the block instance indicated by the dynamic link, at the point indicated by the restart point in the detaching class instance. Before control is transferred the class instance must have its resumption point saved: since it may contain no active sub-blocks, the dynamic link is set to point to the driver itself; the restart pointer to the point in its code following the detach statement (fig. 4).

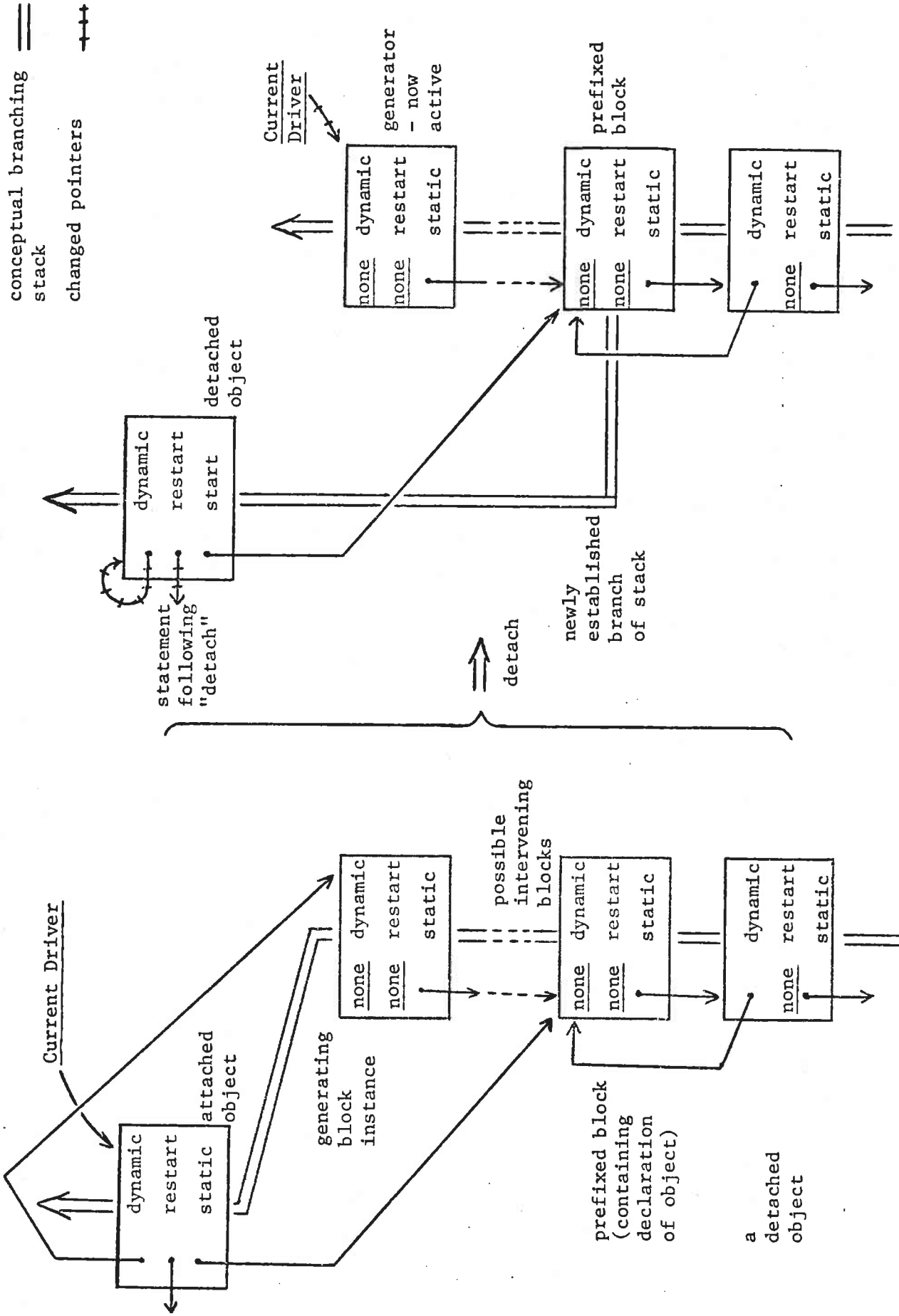


fig. 4. General effect of "detach" from an attached object.

resume:

The structure of the language places restrictions on the classes "visible" from any block, by restricting the possibilities for using prefixed class delarations (see §5). The result which is important here is that in the hierarchy of quasi-parallel systems there is an Operating Chain, which is a chain of drivers (at most one to each quasi-parallel system), starting from the Main Program and connected by dynamic links: and that the dynamically innermost (active) block instance is at the end of this chain: and that that block instance has visible to it only those class instances that are in quasi-parallel systems which have an operating block instance (a member of the Operating Chain) as a member. This implies that a "resume" that has a non-member of the operating chain as destination has to make at most one linkage change in that chain to maintain the discipline, and that change will be in the class instance just one level up from that destination class instance. (See fig. 5.)

(i) The effect of a resume is firstly to save the resumption point of the current object. This object is the smallest (dynamically) enclosing detached object or prefixed block and is found by following dynamic links until a driver of the right kind is met. The resumption point is saved in this driver by setting its dynamic pointer to the current block instance driver, and its restart point to the code after the resume statement.

(ii) The object referred to in the resume statement is used as the first in a chain of dynamic pointers that is followed up through nested quasi-parallel systems (which corresponds to taking the "live" branch at successive nodes upwards through the tree) until a driver with a stored restart point that is not none is reached (which is the base of a sub-branch that branches no further). Its dynamic link indicates the driver that is to be re-activated (at the tip of this branch) and the restart point is a point in that driver's code.

```
begin comment main program;
    begin class A;  begin...end;
      A begin comment outer prefixed block;
        class B;  begin
                  class C;  begin...end;
                  ref (B) tother;
                  detach;
                  C begin comment inner prefixed block;
                    ref (D) done, dtwo;
                    class D;  begin
                                ref (B) getout;
                                detach;
                                   begin
                                   resume (getout);
                                   end;
                              end class D;
                    done :- new D;
                    dtwo :- new D;
                         begin comment sub of prefixed block;
                         done.getout :- tother;
                         resume (tother);
                         resume (done);
                         end;
                    end prefixed block C;
                  end class B;
        bone :- new B;  btwo :- new B;
        bone.tother :- btwo;  btwo.tother :- bone;
                  begin comment sub-block;
                  resume (bone);
                  end;
        end outer prefixed block A;
    end subblock;
end main;
```

fig. 5(a). Code fragment to demonstrate "resume".

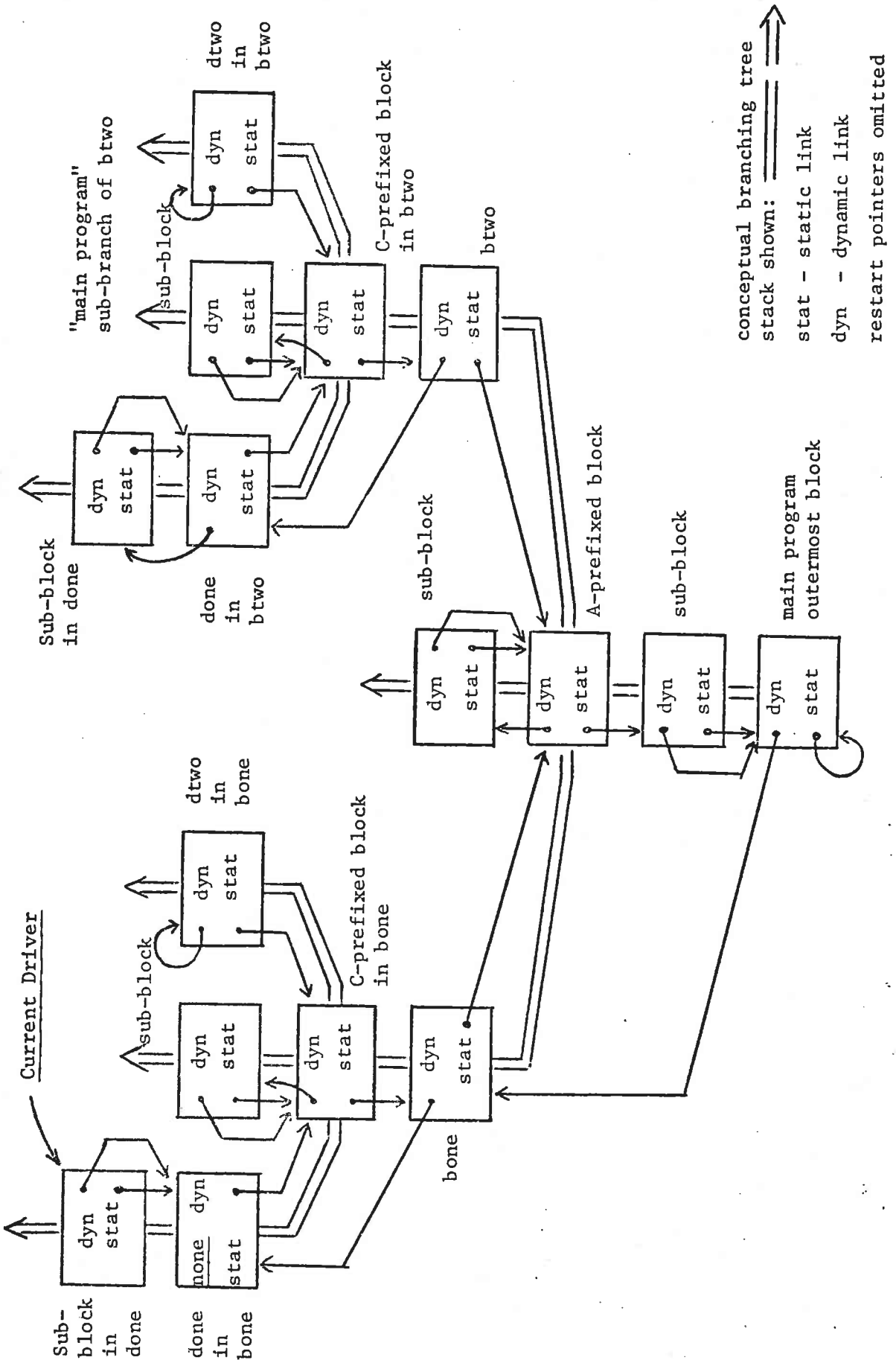


fig. 5(b). Before first "resume (getout)".

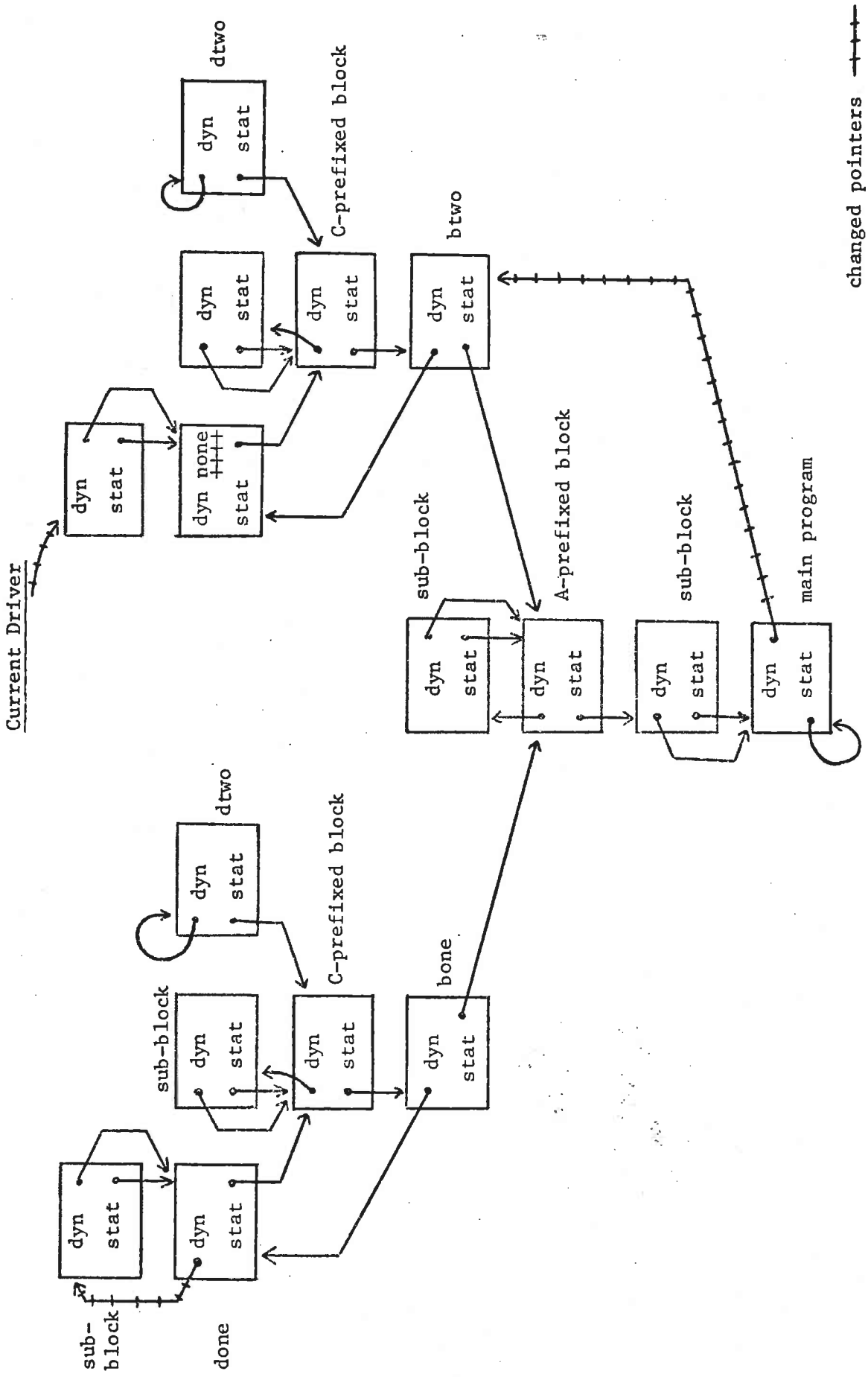


fig. 5(c). After "resume (getout)".

(iii) To maintain the operating chain from the main program tree-base to this point, the static link of detached class instances and the dynamic links of other blocks are followed from the object named in the resume statement to the object which contains its quasi-parallel system. The scoping rules ensure that this object (which is that detached object containing a prefixed block containing the referenced object) is the one whose dynamic pointer needs alteration to maintain the operating chain. It is reset to the object referenced in the resume statement. Control is then passed to the object found in part (ii). (fig. 5)

detach from a detached class instance:

The static link is followed from the Current Driver (which must be that of the detached class instance itself, not one of its sub-blocks, according to the Common Base) and static links followed until a prefixed block is found. This is the smallest enclosing prefixed block. The resumption point is saved in the Current Driver, and dynamic links followed from the block statically outside the prefixed block to find the detached object or prefixed block at the base of the branch supporting this node. Its dynamic pointer is reset to the prefixed block, showing which sub-branch is "alive", and the dynamic pointer in the prefixed block followed to find its resumption point - as in the second part of the operation of "resume", above. (fig. 6)

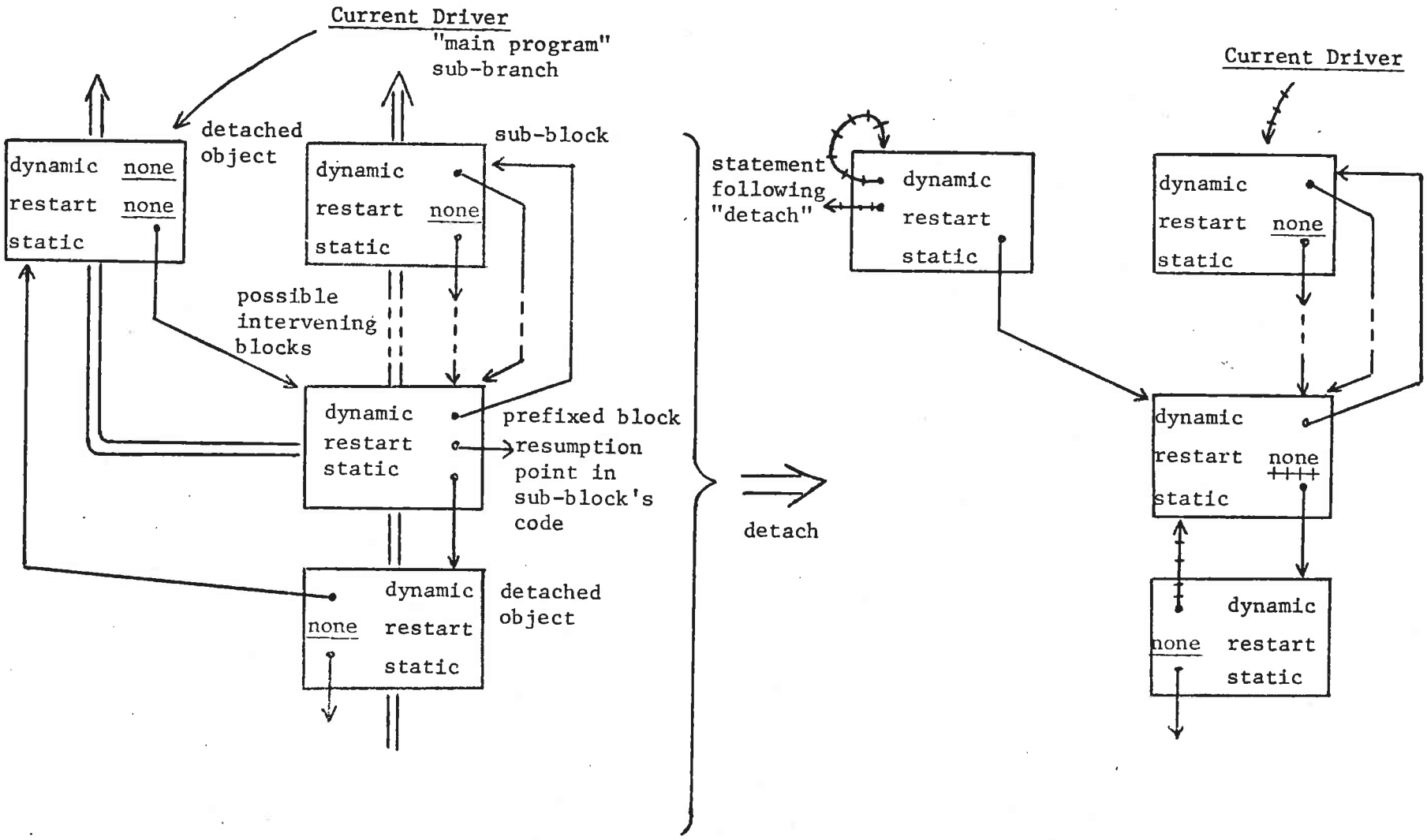


fig. 6. "Detach" from a detached object.

§4. Storage Allocation

Block prototypes (executable code) are stored once and always in a fixed area and make no storage allocation difficulties. The flexible part of storage is divided into two areas - one containing items all of equal size, for drivers: the other containing the varied-sized objects (data areas). Two different systems for storage control of these areas are suggested in the Implementation Guide: one of these is in great detail, with description of the garbage collection algorithms in slightly extended Simula 67, and that scheme will be roughly described here.

In the driver portion of storage free space is chained in driver-sized blocks. Whenever a driver is deleted (on exit from a sub-block, procedure, prefixed block, class body containing no local class declarations; when following a non-local go to; on completion of evaluation of a formal parameter) its storage is returned to the free chain. This process allows complete recovery of such discarded space, but not all deletable drivers fall into these classes. On exit from a prefixed block, for instance, all instances of classes declared within that block can also be deleted - no possible references to them can exist. These instances are not traceable from the prefixed block's driver, and must be detected by searching through all drivers (and objects) to determine those drivers and objects that are still referenceable. This process is quite time-consuming, and is only performed when space in either the driver or object area is exhausted - it is the full garbage collection trace and mark process.

Objects are assigned space in the second area on an increasing address basis - no attempt is made to recover released space until all the storage has been allocated.

Garbage collection is done by a multi-pass algorithm whenever storage in either area is exhausted. Items that must be saved are those that are referenced from the current driver and its associated object, and any object or driver referenced (by dynamic or static environment links or ref variables) by any driver or object that must be saved. All other objects and drivers are discarded and their space recovered. Object space is packed to leave all objects together at the "bottom" of storage and the maximum amount of contiguous storage above. Driver-blocks are moved around in the driver area, but I am unable to discover why: free storage in this area is chained and no fragmentation can occur, and there is no displacement of the boundary between the two areas that would make such movement necessary. Garbage collection is done in several stages to maintain correctness of references between the two areas.

§5. Language Restrictions

Various restrictions have been imposed on the language to make the run-time procedures simpler and eliminate nearly all run-time mode- and reference-checking. The philosophy applied has apparently placed heavy emphasis on security with run-time efficiency and many necessary security checks can be made at compilation time.

The validity of a reference (i.e. whether the referenced object exists, and whether it possesses a quantity as named in a dot expression) is established in nearly all cases at compile time. By requiring refs to be qualified, and restricting the qualification to a class that is visible at the level the ref is declared, it is made impossible to have a ref pointing into limbo (a dangling reference) when the range of the class declaration has been left. If the variable has not yet been assigned a value, it has a default of none - and this condition may be determined with

a very simple check on every ref expression at run-time.

The existence of a named quantity within the referenced object is also ascertainable at compile time, because although a ref qualified by any prefix of a class may be used to refer to instances of that class, the only attributes of the instance accessible via dot-notation with that ref are those that belong to the qualifying prefix class declaration and its prefixes, etc. To access any other attribute, or to make a reference assignment to a ref variable of lower qualification in the prefix hierarchy, an explicit connection clause or qua construction ("instantaneous qualification") must be used. These constructions insert an explicit run-time check for the mode of the actual object, and allow no illegal references to be made: qua gives a run-time error, connection providing an otherwise branch, if the reference is not to an object of the desired class.

A restriction that gets further from the textual scoping concepts of Algol 60 is that classes may only be used as prefixes at the block level at which they are declared - not any inner block level. This restriction ensures that the "resume" procedure at run-time will work reasonably simply - this restriction implying the visibility restrictions mentioned in §3.4 that make maintenance of the operating chain a simple process. It also maintains the validity of references via class prefixes - the range of a class declaration cannot possibly be left before the range of a ref variable qualified by that declaration or one of its prefixes. It also removes possible difficulties with the environment of a prefixed class.

Another restriction removing the need for class checks is: quantities in classes containing local class declarations may not be accessed by dot-notation (see Common Base Language, pp. 55 and 20; Implementation Guide, p.

This restriction also makes the exporting of references from out of their intended context illegal, maintaining the branching stack discipline, and helps ensure that the inner class instance may not still be referenceable when the outer has been lost.

A restriction on the occurrence of "detach" to the outermost block of a class body only is a very broad restriction aimed at preventing the use of "detach" in procedures, which could lead to some unclear situations (as, for instance, when a procedure called as part of an actual parameter expression to a new class instance contains a detach - the class instance can detach before all parameters have been copied, etc.).

§6. Bibliography

Dahl, O.-J. and Myhrhaug, B.

Simula Implementation Guide
Norwegian Computing Centre S47, Oslo (1973).
Contains descriptions of compiler requirements and run-time techniques, in (slightly extended) Simula 67 notation. Has a few errors, and is not easy to understand - no discussion of the basic language and implementation design philosophy is presented.

Dahl, O.-J. Myhrhaug, B., and Nygaard, K.

Common Base Language
Norwegian Computing Centre S22, Oslo (1970).
A formal description of the standard Simula 67, in syntax (BNF) and semantics (English narrative). Not good as an introduction, but the final appeal for reference. The descriptions of the actions of sequencing statements are difficult to follow in this work.

Dahl, O.-J. and Hoare, C.A.R.

Hierarchical Program Structures
in Structured Programming; Dahl, Dijkstra and Hoare, Academic Press, London & New York (1972).
Some features of Simula 67, in examples and description - a high level introduction, assuming knowledge of Algol 60. Some examples do not correspond to the definition of Simula 67.

Birtwistle, G.M., Dahl, O.-J., Myhrhaug, B. and Nygaard, K.

Simula Begin
Studentlitteratur, Lund,Sweden/Auerbach/Philadelphia,Pa. (1973).
Very informal, elementary introduction to Simula 67 as a first programming language. Does not cover a lot of the language features to a useful level of detail.

Wegner, P.

Programming Languages, Information Structures, and Machine Organization
McGraw-Hill, New York (1968).
§4.10.3 for coroutines, §4.6 for the Algol run-time stack.

Lindsey, C.H. and van der Meulen, S.G.

Informal Introduction to ALGOL 68
(Revised Reprint 1973) North Holland/Elsevier/Amsterdam/ London/New York.

Hoare, C.A.R.

Record Handling
in Genuys: Programming Languages, Academic Press (1968).

Jensen, K. and Wirth, N.

Pascal User Manual and Report
Springer-Verlag, New York (2nd ed. 1975).