REFERENCING AND RETENTION IN BLOCK-STRUCTURED COROUTINES

Gary Lindstrom*
Mary Lou Soffa+

UUCS - 79 - 116

November 1979

Department of Computer Science
University of Utah
Salt Lake City, Utah 84112

*Work supported in part by grant MCS-78-03832 from the National Science Foundation to the University of Utah.

+Work supported in part by grant MCS-79-06102 from the National Science Foundation to the University of Pittsburgh.

ABSTRACT

The combination of coroutines with recursive procedures is characteristic of many modern higher-level languages offering advanced control structures (e.g. SIMULA-67, SL5, Interlisp, etc.). We say a language has *block-structured coroutines* (BSCRs) when static nesting considerations govern the usage of this control combination. Starting with the BSCR control description work of Wang and Dahl, this paper pursues further the implications of static program structure on BSCR programs in a compilation-oriented setting. Disciplines on BSCR reference assignment and individual control actions are defined, offering enhanced implementability and program comprehensibility. Of particular interest is a scope-based discipline on "detach" operations, which avoids the formation of idle chain subheads, an implementationally undesirable condition. The retention requirements of BSCRs are analyzed under a range of possible remote accessibility conditions, and two deletion strategies are then defined, keyed to these requirements. The first uses a special form of scope-sensitive reference counting, and the second does mark-sweep garbage collection, again exploiting static program structure. Space and time estimates for both methods are given, along with avenues for continuing research.

1. INTRODUCTION

1.1. Importance of logical parallelism.

Logical parallelism as a programming concept is growing in importance as higher-level programming languages (HLLs) become more widely adopted. Application areas such as systems programming, real time control, simulation, and heuristic searching are but a few of the domains better served by HLLs possessing control structures that go beyond a purely stack-based hierarchy. The term *coroutine*, originally introduced by Conway [6] to characterize the coexistent phase organization often found in compilers, is commonly used to describe such logically parallel program units.

When coupled with recursion in HLLs, coroutines become an extremely flexible tool for algorithm specification. In such a combination recursion contributes dynamic storage allocation and hierarchical environment sharing, while coroutines contribute control versatility within those environments. From these two bases a remarkably wide range of control effects can be achieved, including backtracking [10], pattern-directed invocation [16], and deferred evaluation [9], among many others.

While this versatility is clearly an asset, it is also a liability in that poorly structured control regimes can be encouraged. Thus while languages of this variety possessing purely dynamic control semantics have obvious experimental merit (e.g. SL5 [11]), other languages incorporating coroutines into more compilation-oriented languages with static block structure are likely to have greater ultimate impact. We term this class of languages *block structured coroutine* (BSCR) *languages*. Within this domain, SIMULA-67 [8] offers the most advanced design generally available today.

1.2. Disciplines on BSCR usage.

SIMULA-67 illustrates well the benefits to be obtained by placing disciplines on BSCR usage. These benefits include:

a) implementation economies (such as compile-time reference security);

b) useful semantic concepts offering structure (such as "operating chain", "attachment" vs. "detachment", etc.) in an otherwise amorphous control domain, and

c) convenient control "packages" of higher-level notions (such as the time-based pseudo-parallel control in the system class SIMULATION).

A formal description of SIMULA-67's control behavior (in part) has been offered by Wang and Dahl [23]. However, this early work focused on points (b) and (c) above, and ignored such issues as reference variable scoping rules and the enforcement of control event disciplines. In this paper we address these issues and others, both conceptual and implementational, which arise when BSCR disciplines are chosen especially to exploit program static structure.

1.3. Overview.

We begin by reviewing Wang and Dahl's axiomatic method for describing BSCRs in section 2, along with some clarifying observations on possible BSCR control states. The method is extended in section 3 to include coroutine instance reference variables, constrained to obey certain static scoping rules. The implementation of our selected control event and reference scoping disciplines is treated in section 4. As a prelude to the consideration of BSCR deletion strategies, we present postulates in section 5 that define the retention requirements of our class of BSCRs. In section 6 an incremental deletion strategy $\mathcal{D}'$ is described, based on a new scope-based approach to reference counting. Since $\mathcal{D}'$ is not complete (in that it overlooks certain cyclic structures), a companion method $\mathcal{D}''$ is presented in section 7, based on a form of scope-based mark-sweeping (garbage collection). Section 8 provides a summary and evaluation of our results.

1.4. Previous work.

There have been a wide variety of approaches to the formulation of coroutines; indeed, the title of McIlroy's unpublished but widely circulated memo accurately reflects the situation to this day [17]. Of the few dealing explicitly with scoping issues, the work of Krieg [14] and Vanek [22] are most notable. The latter is of particular interest as a contrast to this work, since in Vanek's approach BSCRs are defined without appeal to programmer manipulatable reference variables.

Approaches to coroutine implementation based on reference counting may be found in [2, 12]; coroutine implementation through garbage collection is studied in [1, 3]. A mixed strategy involving both reference counting and garbage collection may be seen in [4]. Verificational aspects of coroutine programming have been approached in [5, 7].

2. PRELIMINARIES

2.1. Wang and Dahl's approach to control description.

Wang and Dahl approach the description of BSCRs by considering an abstract representation of run-time control states. This representation consists of the set of dynamic procedure[1] *instances* (activations) in existence at any moment, and key relationships that exist among those instances. The relationships of interest are the *dynamic* (i.e. control) and *static* (i.e. textual) connections among the instances.

2.1.1. Primitive symbols. Following the notation of Wang and Dahl, we make use of the following primitive symbols:

S: the set of all procedure instances in existence;

x.SC, for $x \in S$: the *return link* of x, a pair of the form:

[ip: return code pointer, ep: calling instance];

D: a function $S \to S$ denoting dynamic enclosure (*called* $\to$ *caller*), with $D(x) \equiv x.\mathrm{SC}.ep$;

P: the *processor*, in S by extension. By special convention, $P.\mathrm{SC}.ep \equiv D(P) \equiv$ the *currently operating instance*, and $P.\mathrm{SC}.ip \equiv$ the *program counter* of P;

T: a function $S - \{P\} \to S$ denoting static enclosure (*declared* $\to$ *declarer*);

$\to$: a binary relation on S, defined to be $x \to y \equiv x \neq P$ and $D(x) = y$; its transitive closure is $x \overset{+}{\to} y \equiv x \to y$ or $(x \neq P$ and $D(x) \overset{+}{\to} y)$, and its transitive and reflexive closure is $x \overset{*}{\to} y \equiv x = y$ or $(x \neq P$ and $D(x) \overset{*}{\to} y)$.

[1]Blocks in the Algol-60 sense are considered here to be subsumed by the more general notion of procedures. Moreover, since recursion can readily be simulated by LIFO coroutine control, we assume every procedure instance is created as a coroutine.

$\Rightarrow$: a binary relation on S, defined to be $x \Rightarrow y \equiv x \neq P \ \underline{\text{and}}$ $T(x) = y$; its transitive closure is $x \overset{+}{\Rightarrow} y \equiv x \Rightarrow y \ \underline{\text{or}}$ $(x \neq P \ \underline{\text{and}} \ T(x) \overset{+}{\Rightarrow} y)$, and its transitive and reflexive closure is $x \overset{*}{\Rightarrow} y \equiv x = y \ \underline{\text{or}} \ (x \neq P \ \underline{\text{and}} \ T(x) \overset{*}{\Rightarrow} y)$. We denote $\tau(y) \equiv \{ x \mid x \overset{*}{\Rightarrow} y \}$.

OC: the set of instances dynamically linked from $D(P)$, i.e. $\{ y \mid D(P) \overset{*}{\rightarrow} y \}$, is termed the *operating chain* (OC). An instance x is said to be *active* iff $x \in$ OC. If x is active and $x \overset{+}{\rightarrow} y$, we say y is "to the right of" x on the OC.

2.1.2. Control events. Using this notation, we can characterize a class of possible control events. These comprise instance creation (procedure invocation), instance termination, and the coroutine control exchange actions swap(x) and rotate(x, y). Initially, $D(P) = P$.

a) *invocation*: Assume $D(P) = y$ and y invokes a procedure x declared local to an instance z. More precisely, x will refer to a *procedure closure* consisting of a code entry point *code*(x), and a static environment *env*(x), equal to z, providing a data context for accesses of nonlocal variables from within instances of x. Then the result of an invocation of x is the creation of a new instance x of x, adjoined to S, with $T(x) := z$, x.SC := $D(P)$.SC (thereby setting $D(x) = y$), and P.SC := [*code*(x), x].

b) *termination*: Assume $D(P) = x \neq P$. The termination of x causes P.SC to be set to x.SC, and x to be deleted from S.

c) swap(x): This event causes the SC variables of x and P to have their values interchanged, i.e.:

$$\begin{tikzcd} x.\text{SC} \arrow[r, bend right] & P.\text{SC} \arrow[l, bend right] \end{tikzcd}$$

If $x \in$ OC, the effect of swap(x) is a return of control to its caller, without termination of x (i.e., x "detaches"). x.SC saves the *reactivation point* of x.

If $x \not\in$ OC, the effect of swap(x) is the establishment of $D(P)$ as the current caller of x, and the resumption of x at its reactivation point (i.e. x is "called").

d) rotate(x, y): This event causes the SC variables of x, y and P to have their values permuted as indicated below:

$$\begin{tikzcd} & P.\mathrm{SC} \arrow[dl] & \\ x.SC \arrow[rr, bend right] & & y.SC \arrow[ul] \end{tikzcd}$$

The effect of rotate(x, y) is equivalent to swap(x) followed immediately by swap(y) in an indivisible action.

2.2. Selected previous results.

2.2.1. <u>Law and Order</u>. In the interest of semantic coherence and enhanced implementability, Wang and Dahl define four "Law and Order" invariants that circumscribe the range of desirable run-time control states. These are:

(LO1) $P \in S$; the processor is never deleted;

(LO2) D is 1-1; all instances are linked by D into one or more cycles;

(LO3) for all $x \neq P$: $T(x) \in S$; instance deletion observes static nesting order, and

(LO4) for all $x \neq P$: $D(x) \overset{*}{=>} T(x)$; a subtle condition relating D and T link nesting.

2.2.2. <u>Control event conditions</u>. Certain conditions are imposed on control events in order to provide a run-time discipline that will ensure the Law and Order objectives listed above. Assume $D(P) = w$. Then these conditions are:

a) *can invoke* (CI): whenever the language permits w to invoke a procedure x, with env(x) = z, we may be sure that $w \overset{*}{=>} z$; we denote this by w CI x <u>implies</u> $w \overset{*}{=>} env$(x);

b) *can swap* (CS): w CS x implies $w \overset{*}{=>} T(x)$, and

c) *can rotate* (CR): w CR (x, y) implies $w \overset{*}{=>} T(x) \overset{*}{=>} T(y)$.

Wang and Dahl prove that these control event conditions are sufficient to preserve (LO1) - (LO4). The following fact, which will prove useful to us subsequently, is also established:

(LO5) for all w, x: $w \in$ OC and $w \overset{*}{=>} x$ implies $w \overset{*}{\rightarrow} x$.

This condition states that all T links on the OC are local to the OC, and point to the right.

2.2.3. The deletion strategy $\mathcal{D}$. As mentioned in section 2.1.2, instance termination includes the action of deleting that instance from S. Wang and Dahl extend this deletion policy to a more comprehensive policy $\mathcal{D}$ that provides for deletion of instances located off the OC:

$\mathcal{D}$: Assume $D(P) = x \rightarrow y$, $x \neq P$. As the (only) result of the termination of x, P.SC := x.SC (thereby setting $D(P) = y$), and $\tau(x)$ is deleted from S.

$\mathcal{D}$ is then proven to be "safe", in the sense that (LO1) - (LO4) are preserved. One corollary (via (LO2)) is that $\mathcal{D}$ always deletes entire D cycles.

2.3. T link relationships among chains.

Law and Order conditions (LO1) - (LO5) imply a certain structural regularity on T link relationships among chains. We summarize this structure for further reader orientation and to aid in later sections. First, a few additional definitions:

chain(x), $x \in S$: denotes $\{ y \in S \mid x \overset{*}{\rightarrow} y$ or $y \overset{*}{\rightarrow} x \}$; hence chain($P$) = OC;

L: denotes the set of *idle chains*, defined to be $\{$ chain(x) $\mid x \in S$ and chain(x) $\neq$ OC $\}$;

C: denotes the set of all chains, i.e. L union $\{$ OC $\}$;

h(L), L $\in L$: denotes the *head* instance of L, i.e. the object of the swap (or rotate) that created L;

H: denotes the set of all head instances, i.e. $\{ \text{h(L)} \mid \text{L} \in L \}$, and

$=>$: we extend (without confusion) the relation $=>$ on S to apply as well to C. Let L, M be in C. Then L $=>$ M iff L $\neq$ M and there exist $x \in$ L, $y \in$ M such that $x => y$. Similarly, we assume $\overset{+}{=>}$ and $\overset{*}{=>}$ to be extended to C as well.

We now show that $=>$, as is the case on S, imposes a tree structure on C as well.

<u>Theorem 1</u>. T links on OC do not cross. More formally, for all $x, y \in$ OC: $x \overset{*}{\to} y \overset{*}{\to} T(x) \overset{*}{\to} T(y)$ <u>implies</u> $x = y$ <u>or</u> $y = T(x)$ <u>or</u> $T(x) = T(y)$.

<u>Proof</u>. Suppose not. Then there must exist instances x and $y \in$ OC such that $x \overset{+}{\to} y \overset{+}{\to} T(x) \overset{+}{\to} T(y)$; call this property TCross(x, y). Select x and y such that for no z: $x \overset{+}{\to} z \overset{+}{\to} y$, TCross($x$, z) or TCross(z, y). If $x \to y$, (LO4) is clearly violated. Otherwise:

<u>Case 1</u>. For all z such that $x \overset{+}{\to} z \overset{+}{\to} y$, $z \overset{+}{\to} T(z) \overset{*}{\to} y$. Then $D(x) \overset{*}{=>} y$; since $y \overset{+}{\to} T(x) \overset{+}{\to} T(y)$, $D(x) \overset{*}{\not=>} T(x)$, contradicting (LO4).

<u>Case 2</u>. For some z: $x \overset{+}{\to} z \overset{+}{\to} y$, $z \overset{+}{\to} y \overset{+}{\to} T(z)$. But then if $y \overset{+}{\to} T(z) \overset{+}{\to} T(y)$, TCross($z$, y); if $T(y) \overset{*}{\to} T(z)$, TCross($x$, z). Since these are the only two possibilities for $T(z)$ by (LO5), our choice for x and y is contradicted.

<u>Corollary 1</u>. $w \overset{*}{\to} z \overset{*}{\to} x \in$ OC <u>and</u> $w \overset{*}{=>} x$ <u>implies</u> $z \overset{*}{=>} x$.

<u>Corollary 2</u>. Let L be in L. Then (i) T(h(L)) $\notin$ L, and (ii) for all $z \in$ L, $T(z) \in$ L <u>or</u> $T(z) = T$(h(L)).

<u>Proof</u>. (i) Let h(L) $= x$. Then at the creation of L, $D(P) \overset{*}{=>} T(x)$, by CS or CR. Since $x \in$ OC at that time, by (LO5) we have $D(P) \overset{*}{\to} x \overset{+}{\to} T(x)$ Hence $T(x) = T$(h(L)) remains on the OC when L is formed.

(ii) Suppose not. Then for some $z \in$ L, at the creation of L we have $D(P) \overset{*}{\to} z \overset{+}{\to} x \overset{+}{\to} T(z)$, by (LO5). Since $D(P) \overset{*}{=>} T(x)$, by Corollary 1 necessarily $z \overset{*}{=>} T(x)$. Clearly $z \neq T(x)$, so $T(z) \overset{*}{=>} T(x)$. By hypothesis, $T(z) \neq T(x) = T$(h(L)), so $T(z) \overset{+}{=>} T(x)$. Then again by (LO5) we have $z \overset{+}{\to} x \overset{+}{\to} T(z) \overset{+}{\to} T(x)$, contradicting Theorem 1.

Theorem 2. The relation => forms a tree on $\mathcal{C}$.

Proof. Let the node set of the tree be $\mathcal{C}$, and an arc be present from L to M iff L => M. By (LO5), the OC has no outgoing arcs; this is the root of our tree. By Corollary 2, if L ε $\mathcal{L}$ then L => M iff $\mathcal{T}$(h(L)) ε M. Hence the outdegree of L is one, and we have a tree.

3. REFERENCING

3.1. Reference variables.

In the formalism of section 2, there is an implicit assumption that the *names* of procedure instances are directly accessible at the source program level (i.e. for use in swap(x) and rotate(x, y)). Such names, however, are dynamically created and must be manipulated through the use of *reference variables* as intermediaries. We now extend our formalism to deal explicitly with such variables. Our goal here will be to sharpen the control event definitions and conditions of section 2, in preparation for their efficient implementation in sections 4 through 7.

3.1.1. Notation. The following additional notation will be needed:

e: symbols in *lower case script* letters will denote reference variables, i.e. those permitted to assume procedure instance names as values;

$t(e)$: denotes the instance containing the declaration of e as a local variable;

E: (for "environment descriptors") denotes the set of all reference variables currently existing (i.e. local to any existing instance);

$e\uparrow$: denotes the instance currently referenced by e; $e\uparrow$ may also be the special value *nil*, which is the initial value of all $e \in E$;

$stat(x)$: for $x \in S$ denotes the *static depth* of x, defined to be:

$$stat(P) = 0;$$
$$stat(x) = stat(T(x)) + 1 \quad \text{for } x \neq P;$$

$disp(x, i)$: for $x \in S$ and $0 \leq i \leq stat(x)$, denotes the instance statically enclosing x at depth i; that is,

$$T^{stat(x) - i}(x)$$

($disp(D(P), \cdot)$ is the *display* vector of P, used in most block structure implementations for accessing the variables nonlocal to the currently operating instance [19].)

3.1.2. Scoping considerations. We now define a notion of textual scoping for reference variables. In a manner similar to that employed in Algol-68 [21] and Simula-67 [8], we insist that the procedure instance immediately surrounding (statically) an instance x must also surround the declaration of any variable e that references x. We formalize this constraint as follows through a "can reference" (CRF) relation on $E \times S$:

$$e \text{ CRF } x \ \underline{\text{implies}}\ t(e) \overset{*}{\Rightarrow} T(x).$$

By ordinary identifier scoping rules, a relation similar to CRF must exist between the currently operating instance and the instance within which any directly accessible reference variable e is declared. Without confusion, we extend CRF to express this relation on $S \times E$ as well:

$$D(P) \text{ CRF } e \ \underline{\text{implies}}\ D(P) \overset{*}{\Rightarrow} t(e).$$

Note, by transitivity of $\overset{*}{\Rightarrow}$, we have

$$D(P) \text{ CRF } e \text{ CRF } x \ \underline{\text{implies}}\ D(P) \overset{*}{\Rightarrow} T(x).$$

Thus the range of instances directly referenceable from $D(P)$ is a subset of those whose declarations are accessible by ordinary block structure considerations. Moreover, every x such that $D(P) \overset{*}{\Rightarrow} T(x)$ is accessible if by foresight: (i) there are no intervening identifier clashes, and (ii) an e is present at some appropriate block level (e.g. $T(x)$) referencing x.

Instances may be *indirectly* referenced from $D(P)$ through a variety of language structures, including parameter mechanisms and remote accessing (e.g. "e.i" constructs in Simula-67). While instance accessibility through parameters will not be considered here, the full implications of remote accessing will be discussed in sections 5 through 7.

3.1.3. Operations on reference variables. Two primitive operations are defined on E:

a) remref e ("remove reference"), where $e\uparrow = y \in S$ (i.e. $e\uparrow \neq nil$): the reference to y is cleared from e, and nil is installed in its place.

b) $\mathcal{e}$ setref α ("set reference"), where $\mathcal{e}\uparrow = nil$ and α is an expression evaluating to the name of some $y \in S$: if $\mathcal{e}$ CRF y, then $\mathcal{e}$ is made to point to y (otherwise, an error occurs).

Note that for our later implementational convenience we assume remref is never done when $\mathcal{e}\uparrow = nil$; similarly, $\mathcal{e}$ setref α is done only when $\mathcal{e}\uparrow = nil$ and α has a non-nil value. Thus an arbitrary statement of the form $\mathcal{e} := \alpha$ would correspond to the code sequence:

```
if e↑ ≠ nil then remref e;
if α ≠ nil then e setref α.
```

3.2. Control events with referencing.

3.2.1. Extended definitions. To support the mechanics of referencing, we choose to make the following changes to the specifications of section 2.1.2:

a) *invocation*: Assume $D(P) = y$ and y has access to a procedure closure $x = [\ code(x),\ env(x)\ ]$. Then an invocation of x, denoted new x, causes the following actions: a new instance x of x is created (adjoined to S), $T(x) := env(x)$, x.SC $:= [\ code(x),\ x\ ]$, and the operator new returns x as its value. (A typical application would be of the form $\mathcal{e}$ setref new x.) Note that in contrast to 2.1.2.a, the execution of x does not commence, since P.SC is unchanged; instead, $L := L$ union $\{\ x\ \}$.

b) *termination*: Assume $D(P) = y$ and y terminates. Then a detach(y) is done (see (c) below), and y.SC.ip := *undefined*. Note that y is not automatically deleted; this will be handled more comprehensively in sections 6 and 7. For our later convenience we denote $T = \{\ y \in S \mid y.\text{SC}.ip = undefined\ \}$.

c) swap : As indicated in 2.1.2.c, swap is actually a combined specification of two distinct control events, coroutine detachment and reactivation (call). In the interest of program clarity and implementability, we follow the lead of Simula-67 and denote each case separately. Let α be an expression evaluating to some instance y. Then:

call(α), for $y \notin$ OC: y.SC and P.SC have their values interchanged. Chain(y) is thereby removed from L, and appended to the left end of the OC.

detach(α), for $y \in$ OC: y.SC and P.SC have their values interchanged. Chain(y) is thereby created, and adjoined to L.

d) rotate(α, β): equivalent to { detach(α); call(β) }.

3.2.2. Extended conditions. Our revised control events do not require any changes to be made to the CI condition of 2.2.2. However, some modifications are necessary to CS and CR. Moreover, we choose to build in at this time an added constraint on idle chain reactivations. It is programmatically important that dynamic enclosure relationships among instances within an idle chain be preserved over that chain's creation and reactivation. That is, suppose $x \overset{+}{\rightarrow} y$ on the OC, and $x \in$ chain(y) = L $\in L$ after a detach (or rotate), i.e. x and y are idled together. Then upon reactivation of L we wish $x \overset{+}{\rightarrow} y$ to hold once more, rather than $y \overset{+}{\rightarrow} x$. This policy may be ensured by constraining call to take only arguments which are members of H, i.e. idle chain head instances.

a) *can detach* (CD): Suppose detach(α) is to be executed. Then necessarily:

(CD1) α must evaluate to some $y \in S$ (hence is non-*nil* in value;

(CD2) $D(P) \overset{*}{\Rightarrow} T(y)$, and

(CD3) $y \in$ OC.

b) *can call* (CC): Suppose call(α) is to be executed. Then necessarily:

(CC1) α evaluates to some $y \in S$;

(CC2) $D(P) \overset{*}{=>} T(y)$;

(CC3) $y \in H$, and

(CC4) $y \not\in T$.

c) *can rotate* (CR): Suppose rotate(α, β) is to be executed. Then necessarily:

(CR1) α evaluates to some $x \in S$;

(CR2) β evaluates to some $y \in S$;

(CR3) $D(P) \overset{*}{=>} T(x) \overset{*}{=>} T(y)$;

(CR4) $x \in$ OC;

(CR5) $y \in H$, and

(CR6) $y \not\in T$.

4. IMPLEMENTING REFERENCING

The conditions described in section 3 may be implemented efficiently using a combination of compile-time and run-time checks. We consider each condition in turn.

4.1. CRF.

The referenceability relation CRF has two aspects, as specified in section 3.1.2.

4.1.1. $\underline{E} \underline{\times} \underline{S}$. The condition e CRF y, i.e. $t(e) \overset{*}{\Rightarrow} T(y)$, may be checked at run-time by verifying that $stat(t(e)) \geq stat(y) - 1$, and that $disp(t(e), stat(y) - 1) = T(y)$. Note that if $D(P)$ CRF e (e.g. e is being accessed directly), the test can be made in fixed time through use of P's display vector. Otherwise, this test will require $stat(t(e)) - stat(y) + 1$ steps along the T link sequence from $t(e)$.[2]

4.1.2. $\underline{S} \underline{\times} \underline{E}$. The condition $D(P)$ CRF e, i.e. $D(P) \overset{*}{\Rightarrow} t(e)$, may be compile-time checked as per ordinary identifier scoping rules.

4.2. CC and CD.

The verification of conditions (CD1) and (CC1) require simply a non-*nil* value check at run-time. Conditions (CD2) and (CC2) can be accomplished as per section 4.1.1. Condition (CC4) is trivial. Conditions (CD3) and (CC3), however, require some elaboration. We assume an H membership bit in each instance x, denoted Hbit(x), set by new and detach, and cleared by call.

4.2.1. <u>(CC3)</u>. We implement the test for $y \in H$ as a run-time test for Hbit(y) = 1.

4.2.2. <u>(CD3)</u>. Implementing the test for $y \in$ OC is more challenging, since Hbit(y) = 0 for both $y \in$ OC and $y \in$ L $\in$ L, where $y \neq$ h(chain(y)). Stepwise traversal of the OC looking for y is unattractive, as is the idea of an OC membership bit. A better approach exists exploiting Theorem 1 and the fact that (CD3) is only tested when $D(P) \overset{*}{\Rightarrow} T(y)$.

[2]In the simple case $e:=f$, where e and f are accessed directly, no run time CRF tests are necessary if $stat(t(f)) \leq stat(t(e))$.

Suppose $D(P) \overset{*}{\Rightarrow} T(y)$ and $y \in$ OC is to be verified. Consider each $y_i \in$ OC such that $T(y_i) = T(y)$, numbered from left to right. By (LO5), we have $D(P) \overset{*}{\to} y_1 \overset{+}{\to} \dots \overset{+}{\to} y_k \overset{+}{\to} T(y)$. If $y \in$ OC, $y = y_i$ for some $1 \leq i \leq k$. By Corollary 1 of Theorem 1, we find each y_i as follows. Let b = $stat(y)$. Then $y_1 = disp(D(P), \mathrm{b})$, $y_{i+1} = disp(D(y_i), \mathrm{b})$ for $1 \leq i \leq k-1$, and $D(y_k) = T(y)$. Hence:

```
isonOC($y$):  { $test\ for\ y \in OC,\ given\ D(P) \overset{*}{\Rightarrow} T(y)$ }
begin $z$ := $D(P)$;  foundy := false;
    while $z \neq T(y)$ and not foundy do
    begin $z$ := $disp(z, b)$;
        foundy := $z = y$;
        $z$ := $D(z)$
    end;
    isonOC := foundy
end
```

4.3. CR.

Conditions (CR1-2) and (CR4-6) correspond directly to similar conditions discussed in section 4.2. We consider now condition (CR3), $D(P) \overset{*}{\Rightarrow} T(x) \overset{*}{\Rightarrow} T(y)$. In a manner analogous to that for (CC2) and (CD2), we implement (CR3) by the following run-time checks:

(i) $stat(D(P)) \geq stat(x) - 1 \geq stat(y) - 1$;

(ii) $disp(D(P), stat(x) - 1) = T(x)$, and

(iii) $disp(D(P), stat(y) - 1) = T(y)$.

We observe that all these tests can be accomplished in fixed time using P's display vector. Moreover, we note that in the special case where rotate(e, f) is being done on two reference variables directly accessible from $D(P)$, steps (ii) and (iii) may be eliminated.

Theorem 3. Suppose $D(P)$ CRF e and $D(P)$ CRF f, with $e\uparrow = x \in S$ and $f\uparrow = y \in S$. Then $D(P) \overset{*}{\Rightarrow} T(x) \overset{*}{\Rightarrow} T(y)$ iff $stat(x) \geq stat(y)$.

Proof. Clearly $T(x) \overset{*}{\Rightarrow} T(y)$ implies $stat(x) \geq stat(y)$. Now assume the latter. By CRF, $D(P) \overset{*}{\Rightarrow} t(e) \overset{*}{\Rightarrow} T(x)$ and $D(P) \overset{*}{\Rightarrow} t(f) \overset{*}{\Rightarrow} T(y)$. Now either $D(P) \overset{*}{\Rightarrow} T(x) \overset{*}{\Rightarrow} T(y)$, and we are done, or $D(P) \overset{*}{\Rightarrow} T(y) \overset{+}{\Rightarrow} T(x)$. The latter contradicts our assumption of $stat(x) \geq stat(y)$.

4.4. Idle chain subheads.

As discussed in section 4.2, Hbit(x) may be used to detect immediately if $x \in H$ but not if $x \in$ OC, even given $D(P) \overset{*}{=>} T(x)$. This difficulty is caused by the possible presence of idle chain *subheads*, which we now define.

Definition. Suppose $x \in$ L $\in \mathcal{L}$. Then x is said to be a *subhead* of L iff $x \notin H$ and $T(x) \notin$ L. By Corollary 2 of Theorem 1, $T(x) = T$(h(L)).

4.4.1. Subhead absence. We now argue that the absence of idle chain subheads improves the implementability of (CD3). Further benefits will become evident in sections 6 and 7.

Theorem 4. If no subheads exist on chains in $\mathcal{L}$, then (CD3) may be implemented as an Hbit(y) = 0 test.

Proof. Clearly, if Hbit(y) = 1, $y \in H$, so $y \notin$ OC. Now suppose $D(P) \overset{*}{=>} T(y)$ and Hbit(y) = 0. Then either $y \in$ OC, and we are done, or $y \in$ chain(y) $\in \mathcal{L}$. Since $T(y) \in$ OC, y must be either in H or be a subhead of chain(y). Both are impossible by assumption.

4.4.2. Subhead creation. Idle chains with subheads are created when detaches are done on instances not statically surrounding the currently operating instance.

Theorem 5. Suppose $D(P) = w \overset{*}{=>} T(y)$ and a detach is done on y. Then chain(y) has no subheads iff $w \overset{*}{=>} y$.

Proof. Assume chain(y) is formed with no subheads. Then $y \in H$ and for all $z \in$ chain(y), $z \overset{*}{=>} y$. In particular, $w \overset{*}{=>} y$.

Now assume $w \overset{*}{=>} y$. Consider any z such that $w \overset{*}{\rightarrow} z \overset{*}{\rightarrow} y$. By Corollary 1 of Theorem 1, $z \overset{*}{=>} y$. Hence chain(y) has no subheads.

4.4.3. Preventing subheads. By Theorem 5, idle chain subheads can be avoided, with no other loss of flexibility, by enforcing (CD3') $D(P) \overset{*}{=>} y$ rather than (CD3) $D(P) \overset{*}{=>} T(y)$. This may be achieved, if desired, in one of two ways:

a) By adopting (CD3'), with implementation $stat(y) \leq stat(D(P))$ and $disp(D(P), stat(y)) = y$, or

b) By expressing detaches in terms of surrounding block labels, e.g.:

detach(*<procedure identifier>*), and

rotate(*<procedure identifier>*, *<reference expression>*).

Option (b) is not only more compilable and avoids awkward run-time errors, but also yields a more readable, statically comprehensible program text. Similar conventions are now being advocated for exception handling in block structured languages.

5. RETENTION REQUIREMENTS

In preparation for our discussion of reference-based instance deletion strategies, we assess the retention requirements imposed by the referencing and control event conditions described in section 3.

5.1. Instance accessibility.

Instances must be retained as long as their presence may be instrumental to the program's continued execution. Clearly, all instances which may become active must be retained, as must instances containing variables that may be accessed from potentially active instances. Such variables are said to be *remotely accessed.*

To illustrate, assume that $e\uparrow = x$, and that $\mathtt{i}$ is an identifier. Let us interpret the notation $e.\mathtt{i}$ as denoting the variable declared local to x under $\mathtt{i}$ (assuming it exists). Then at least three policies on the legality of $e.\mathtt{i}$ are possible:

<u>Policy</u> (<u>a</u>): <u>no</u> remote accessibility is permitted, i.e. constructs of the form $e.\mathtt{i}$ are prohibited;

<u>Policy</u> (<u>b</u>): <u>single-level</u> remote accessibility is permitted, i.e. $e.\mathtt{i}$ is permitted if the variable denoted is not a reference variable, and

<u>Policy</u> (<u>c</u>): <u>multi-level</u> remote accessibility is permitted, i.e. $e.\mathtt{i}.\mathtt{f}$ is permitted if $e.\mathtt{i}$ denotes a reference variable and $e.\mathtt{i}.\mathtt{f}$ exists, etc.

Clearly, a severe remote accessing policy such as (a) presents earlier opportunities for deletion at the cost of programming flexibility. Policy (c) weights these two features in an opposite fashion. Policy (b) provides an attractive balance, offering certain programming conveniences (e.g. the use of terminated instances as simple data records), while permitting reasonable timeliness in instance deletion.

5.2. The sets Act and Ret.

We may formally define the retention requirements in our class of BSCR languages with the aid of two subsets of S:

Act: the set of instances which at present are not deletable, due to their potential activatability (entrance onto OC), and

Ret: the set of instances which at present are either potentially activatable, or are referenceable from a potentially activatable instance.

By inspection of the control events of section 3 and the remote accessing policies above, we may define Act and Ret to be the smallest subsets of S satisfying the following postulates:

(AR1) Act is a subset of Ret;

(AR2) OC is a subset of Act;

(AR3) $x \in H$ <u>and</u> $x \in Act$ <u>implies</u> chain(x) is a subset of Act;

(AR4) Under remote accessing policies (a) and (b):

$$t(e) \in Act \ \underline{\text{and}}\ e\uparrow \neq nil \ \underline{\text{implies}}\ e\uparrow \in Ret;$$

under policy (c):

$$t(e) \in Ret \ \underline{\text{and}}\ e\uparrow \neq nil \ \underline{\text{implies}}\ e\uparrow \in Ret;$$

(AR5) $x \in Ret$ <u>and</u> $x \in H$ <u>and</u> $x \not\in T$ <u>and</u> $T(x) \in Act$ <u>implies</u> $x \in Act$.

Conditions (AR1-3) are obvious, and (AR4) follows directly from the definition of our remote accessing policies. Condition (AR5) embodies (CC3), the relevant condition on call, which is the only means by which instances can become active.

5.3. Instance deletion and termination.

We now clarify the meaning of instance deletion and termination, and define the applicability of such actions in terms of Ret and Act.

Definition. The *deletion* of an instance x involves its removal from S (i.e., the availability of its storage for reuse). The *termination* of x involves x.SC := [*undefined*, x], and Hbit(x) := 1 (i.e., x is made into a singleton idle chain that is no longer callable).

Under policy (a), we may at any time delete instances in S - *Act*.

Under policies (b) and (c), we may at any time delete instances in S - *Ret*, and terminate instances in *Ret* - *Act* - T (i.e., make T = *Ret* - *Act*).

As mentioned in section 5.1, policy (a) clearly does present greater opportunity for instance deletion. However, an "aggressive" deletion strategy under policy (a) would pose the problem of either locating (and clearing) all references to an instance in *Ret* - *Act* prior to its deletion, or dealing securely with the problem of "obsolete" references to potentially recycled storage. For this pragmatic reason, we will assume henceforth that policy (a) is implemented as a variation of policy (b), with remote accessing prohibited.

5.4. Properties of *Ret* and *Act*.

Clearly, the construction of *Ret* and *Act* can be done through any order of application of (AR1-5). For our later convenience in inductive proofs, we will assume that instances enter *Ret* and *Act* one by one, and that the following ordering is observed:

a) (AR1) has top priority (i.e. if x enters *Act* - *Ret*, then it enters *Ret* without delay), and

b) when (AR2) or (AR3) are applied, the elements of the chain enter *Act* (and, by (a), *Ret*) in right to left order.

Given this standard construction method, we have:

(AR6) If x enters Act, $x = P$ or $T(x) \in Act$ already.

Proof. By (b) above, this order will be observed within chains. An instance x with $T(x) \not\subset \text{chain}(x)$ enters Act only by (AR5), with $T(x) \in Act$ already by requirement.

(AR7) If x enters Ret, $x = P$ or $T(x) \in Ret$ already.

Proof. We proceed by induction on the assumed construction order of Ret and Act. If $x \neq P$ enters Ret by (AR1), $T(x) \in Act$ already by (AR6); hence $T(x) \in Ret$ already by assumption (a). Otherwise, x must enter Ret by (AR4). No matter which policy is in effect, $t(e) \in Ret$ already. By CRF, $t(e) \overset{*}{=>} T(x)$, so by induction $T(x) \in Ret$ already.

Given (AR6) and (AR7), we can prove the following useful fact.

Theorem 6. If policy (b) (or (a)) is in effect, then (AR4-5) may be combined as follows:

(AR4') $t(e) \in Act$ and $e\uparrow \neq nil$ implies $e\uparrow \in Ret$ and ($e\uparrow \in H$ and $e\uparrow \not\subset T$ implies $e\uparrow \in Act$).

Proof. Since (AR4') includes (AR4), we need only check that (AR4') subsumes (AR5) as well. (AR5) is crucial to instances that enter Act by virtue of first being in Ret. Instances enter $Ret - Act$ by (AR4). On such an occasion we have $t(e) \in Act$, with $t(e) \overset{*}{=>} T(e\uparrow)$ by CRF. By (AR6), we are sure $T(e\uparrow) \in Act$, so the remaining conditions for (AR5) may be incorporated directly into (AR4').

6. SCOPE-BASED REFERENCE COUNTING

6.1. Overview.

Wang and Dahl's deletion strategy $\mathcal{D}$ (section 2.2.3) provides for the deletion of $\tau(x)$ upon the termination of x. With the introduction of referencing, $\mathcal{D}$ is no longer appropriate because:

a) terminated instances may no longer be automatically deleted if remote referencing is permitted, and

b) other opportunities for deletion can be occasioned by remref operations and detaches that are *de facto* terminations due to loss of instance accessibility.

In this section we define a new deletion strategy $\mathcal{D}'$ exploiting a form of scope-based reference counting. While $\mathcal{D}'$ is economical and incremental, it is not complete in the sense that some deletion opportunities may be overlooked (namely, those involving circular references of a particular form). Section 7 deals with a companion strategy $\mathcal{D}''$ that ameliorates this shortcoming through a form of scope-based garbage collection.

6.2. *Ext* fields.

References to an instance x from outside $\tau(x)$ have a greater retentive influence on x than do those from within $\tau(x)$. For this reason, we choose to count such instances in a field $ext(x)$. Thus:

$$ext(x) = |\{ e \mid e\uparrow = x \ \underline{\text{and}}\ t(e) \overset{*}{\not>} x \}|$$

6.2.1. Maintaining *ext* fields. For each $x \in S$, $ext(x)$ may be maintained economically as follows:

i) new: $ext(x) := 0$ upon creation of x;

ii) e setref α, where α evaluates to x:
   if $stat(t(e)) < stat(x)$ or $disp(t(e), stat(x)) \neq x$
   then $ext(x) := ext(x) + 1$, and

iii) remref e, where $e\uparrow = x$:
   if $stat(t(e)) < stat(x)$ or $disp(t(e), stat(x)) \neq x$
   then $ext(x) := ext(x) - 1$.

Again we observe that if $D(P)$ CRF $\mathcal{e}$, actions (ii) and (iii) each take fixed time through the use of P's display vector.

6.2.2. Exploiting ext counts. The utility of ext counts in detecting deletion and termination opportunities may be seen by the following theorems.

Theorem 7. $x \in H$ and $ext(x) = 0$ implies $x \notin \mathcal{Ret}$.

Proof. Since $P \notin H$, the proposition is trivially true for $x = P$. We proceed by induction on the assumed construction order of $\mathcal{Ret}$ and $\mathcal{Act}$. Consider the moment when x enters $\mathcal{Ret}$. If it enters by (AR3) and (AR1), $x \notin H$. Otherwise, it must enter by (AR4), with $t(\mathcal{e}) \in \mathcal{Ret}$ already and $\mathcal{e}^{\uparrow} = x$. If $t(\mathcal{e}) \overset{*}{\not\Rightarrow} x$, $ext(x) > 0$ because of $\mathcal{e}$. Otherwise, $t(\mathcal{e}) \overset{*}{\Rightarrow} x$, so $x \in \mathcal{Ret}$ already by (AR7), a contradiction.

Corollary 1. If $x \in H$ and $ext(x) = 0$, $\tau(x)$ and $\mathcal{Ret}$ are disjoint.

Proof. Follows directly from Theorem 7 and (AR7).

Corollary 2. If $x \in H$ and $ext(x) = 0$, chain(x) and $\mathcal{Act}$ are disjoint.

Proof. By Theorem 7, $x \notin \mathcal{Ret}$, hence $x \notin \mathcal{Act}$. By (AR3) and (AR5), chain(x) and $\mathcal{Act}$ are then necessarily disjoint.

Corollary 3. If $x \in H$ and $ext(x) = 0$, $\tau(x)$ may be deleted and for each subhead $y \in$ chain(x), $\tau(y)$ may be deleted if $ext(y) = 0$ and terminated otherwise.

Proof. By Corollary 1, $\tau(x)$ is disjoint from $\mathcal{Ret}$ and may be deleted. By Corollary 2, $y \notin \mathcal{Act}$ for each subhead $y \in$ chain(x). Hence $\tau(y)$ may be terminated. If $ext(y) = 0$, $\tau(y)$ may then be deleted since y will then be a head instance (of a singleton chain).

6.3. $\mathcal{D}'$ invariants.

Before describing the operation of $\mathcal{D}'$ in detail, we specify its desired behavior through a list of invariants that are to be preserved (in addition to (LO1-4)):

(RC1) for all $e \in E$, $e\uparrow = nil$ <u>or</u> $e\uparrow \in S$.

(RC2) for all $x \in S$, $ext(x) = |\{ e \in E \mid t(e) \overset{*}{\not=>} x \}|$.

(RC3) for all $L \in L$, $ext(h(L)) > 0$.

(RC4) for all $x \in T$,

a) $D(x) = x$;

b) $\tau(x)$ is a subset of T, and

c) if policy (b) is in effect, for all $e \in E$ such that $t(e) = x$, $e\uparrow = nil$.

6.4. Opportunities under D'.

Given properly maintained *ext* count fields, the instance deletion and termination opportunities provided by Theorem 7 and its corollaries may be exploited as follows.

6.4.1. <u>Detection</u>. There are three occasions upon which the conditions for D' can apply.

i) remref e, with $e\uparrow = x$: if $ext(x)$ is brought to zero and $x \in H$, we may delete $\tau(x)$ and terminate $\tau(y)$ for each subhead y in chain(x).

ii) detach α, with α evaluating to x: if $ext(x) = 0$, we may again delete $\tau(x)$ and terminate $\tau(y)$ for each subhead y in the newly created chain(x).

iii) terminate, with $D(P) = x$: if $ext(x) = 0$, we may delete $\tau(x)$; otherwise, we may terminate $\tau(x)$.

6.4.2. <u>Effects</u>. The reference counting assumed under D' requires certain effects in instance deletion and termination beyond those defined in section 5.3. We sharpen these specifications here, in preparation for their algorithmic accomplishment in section 6.5.

i) $\tau(x)$ *deletion*:

a) for all e such that $t(e) \overset{*}{=>} x$, if $e\uparrow \neq nil$ and $e\uparrow \overset{*}{\not=>} x$, remref e.

b) for all $y \in \tau(x)$, remove y from S.

ii) $\tau(x)$ *termination*:

a) if $ext(x) = 0$, delete $\tau(x)$. Otherwise:

b) Under policy (b):

1) for all e such that $t(e) = x$, remref e;
2) for all y such that $T(y) = x$, delete $\tau(y)$, and
3) terminate x.

c) Under policy (c):

1) for all $y \notin T$ such that $T(y) = x$, terminate $\tau(y)$, and
2) terminate x.

6.5. Implementing $\mathcal{D}'$.

The processes of τ-tree deletion and termination specified above are highly recursive. Not only are they *statically* recursive in traversing $\tau(x)$, but also *dynamically* recursive due to *rippling* effects. That is, the deletion of $\tau(x)$ may cause the deletion (or termination) of a disjoint $\tau(y)$ to be triggered through a remref operation within $\tau(x)$. Moreover, the termination of $\tau(x)$ can cause under policy (c) the cascaded deletion of several subtrees of $\tau(x)$ in unpredictable order. These effects can cause not only bookkeeping problems but uncontrolled space requirements if not carefully implemented.

6.5.1. The worklist W. For these reasons we implement τ-tree deletion and termination as cooperating algorithms processing a queue of instances needing their attention. We term that queue the *worklist* W. Fortunately, all such needy instances can relinquish their previous chain membership, so we can implement W as a special pseudo-chain, with its front ("head") pointed to by *Wfront*, and its rear by *D(Wfront)*. Then:

i) if $x \in W$ with $ext(x) = 0$, $\tau(x)$ is to be deleted; otherwise, $\tau(x)$ is to be terminated.

ii) we assume W is sorted such that $stat(x) \geq stat(D(x))$ for $x \in W$ and x not at the front of W.

6.5.2. Seeding W. Instances enter W through one of the three occasions specified in section 6.4.1. By examination of these conditions, we observe that one common algorithm suffices: queuechain (fig. 1). We assume that each such program event causes a queuechain invocation, followed by processqueue. Rippled remrefs may cause further W loading (via schedule), but these are handled in due course by the same continuing processqueue execution.

6.5.3. Representing $\tau(x)$. We assume each instance x has two link fields *desc* and *sib*, representing $\tau(x)$ as follows. Suppose x has k descendants y_i, each such that $T(y_i) = x$. Then:

$$sib^{i-1}(desc(x)) = y_i,\ 1 \leq i \leq k, \text{ and}$$
$$sib^{k}(desc(x)) = nil.$$

While this unidirectional linking of siblings will occasionally cause inefficiency when "random" deletions are to be done, such occurrences will be kept to a minimum. The alternative is bidirectional linking, which we judge to be unnecessary.

6.5.4. τ-tree deletion. The implementation of deletetree(x) is given in fig. 2. We make the following observations:

i) Only x itself within $\tau(x)$ requires a random deletion from a sibling list.

ii) The two pass nature of deletedesc is required so that accesses of $stat(e\uparrow)$ do not occur after $e\uparrow$ has been destroyed. If each e carries $stat(e\uparrow)$ as part of its value, or if one may be sure the space possessed by instance $e\uparrow$ cannot immediately be reallocated, the two passes can be merged.

iii) For efficiency, we implement deletedesc recursively. The depth of recursion is limited by the program's maximum static depth, and therefore represents a known space requirement.

iv) Since W is sorted by nonincreasing *stat* order, W and $\tau(x)$ are disjoint. Hence we are free to delete any $y \in \tau(x)$ without affecting W.

```
queuechain(x):   {x a head instance; x and subheads
                        of chain(x) are to be inserted into W}

begin local y, z;
      z := D(x);  {make z point to left end of chain(x)}
      schedule(x);  {sort x into W}
      loop  y := disp(z, stat(x));  {y := next subhead (or x)}
      exit if y=x;
            z := D(y);
            D(y) := D(x);  D(x) := y  {insert y in front of x on W}
      end;
      Hbit(x) := 0  {prevents rescheduling of x}
end


processqueue:   {process W doing tree deletions and/or termina-
                        tions until W is empty}

begin local y;
      while Wfront≠nil do
      begin y := D(Wfront);  {remove leftmost element of W}
            if y=Wfront then Wfront:=nil
            else D(Wfront) := D(y);
            if ext(y)=0 then
                  deletetree(y)
            else  terminatetree(y)
      end
end
```

Fig. 1. $\mathcal{D}'$ worklist management (schedule code omitted).

```
deletetree(x):    {delete all y such that $y \overset{*}{\Rightarrow} x$}

begin deletedesc(x, x); {process descendants of x}
      remove(x, T(x));   {do "random" removal of x as desc. of T(x)}
      destroy(x)   {reclaim x's storage}
end


deletedesc(z, x):    {given $z \overset{*}{\Rightarrow} x$, clear all refs in $\tau(z)$ departing
                      $\tau(x)$, and destroy $\tau(z)$ except z;
                      all y referenceable from $\tau(z)$ still exist}
begin local w, y;
      clearrefs(z, x);
      y := desc(z);
      while y≠nil do  {process descendants of z}
      begin deletedesc(y, x);  y:=sib(y)
      end;
      y := desc(z);
      while y≠nil do  {destroy immediate descendants of z}
      begin w := sib(y);  destroy(y);
            y := w
      end;
      desc(z) := nil
end


clearrefs(z, x):   {given $z \overset{*}{\Rightarrow} x$, clear all refs in z departing $\tau(x)$}

begin local e;
      for each e such that t(e)=z do
            if e↑≠nil then
                  if stat(e↑)≤stat(x) and e↑≠x then remref e
end
```

Fig. 2. Tree deletion within $\mathcal{D}'$ (remove and destroy code omitted).

v) Since queuechain is invoked only on idle chain heads, and elements of W are no longer in H, any instance scheduled via a rippled remref action must currently be unscheduled.

vi) A $\tau(x)$ scheduled for termination may in fact be ready for deletion when reached by processqueue if $ext(x) = 0$ by that time. If so, deletion occurs rather than termination.

6.5.5. Tree termination. In contrast to deletetree, terminatetree(x) can cause subtree deletions within $\tau(x)$ in unpredictable order (fig. 3). For this reason, we exploit the worklist W to process such rippling effects rather than through recursion. Again, the static depth order imposed on W is crucial, so that we neither delete instances while scheduled, nor schedule them redundantly (thereby malforming W).

6.5.6. Efficiency. As observed above, the space required for the operation of $\mathcal{D}'$ is proportional to the maximum program static depth. This remains true even if for speed purposes we maintain a *Wdisplay* vector permitting fixed-time insertion into W.

The time required under $\mathcal{D}'$ is linear with respect to the number of instances deleted or terminated, except for:

i) occasional random removals from descendant lists (remove in deletetree), and

ii) the examination (once) of each of the previously terminated immediate descendants of each newly terminated instance.

```
terminatetree(x):   {terminate τ(x)}

begin local y;
      term(x);  {terminate x itself}
      y := desc(x);  {nil if policy(b) holds}
      while y≠nil do   {use W for desc. needing term./del.}
      begin if ext(y)=0 or y.SC.ip≠undefined then
                  schedule(y);
            y := sib(y)
      end
end


term(x):   {terminate x, i.e. make into non-CALLable singleton chain}

begin x.SC.ip := undefined;
      D(x) := x;
      Hbit(x) := 1;
      if policy(b) then  {only x need be retained}
            deletedesc(x, x)
end
```

Fig. 3. Tree termination in $\mathcal{D}'$.

7. SCOPE-BASED GARBAGE COLLECTION

7.1. Overview.

Deletion strategy $\mathcal{D}'$ just described offers incremental storage reclamation at reasonable space and time cost. If sibling reference cycles are not created, or if the parents of such siblings are deleted (or terminated under policy (b)) before space exhaustion occurs, $\mathcal{D}'$ suffices permanently. However any complete implementation of the control forms under discussion must provide a back-up mechanism for detecting such cycles and deleting them through a thorough search for sets Ret and Act.

In this section we present $\mathcal{D}''$, a mark-sweep or garbage collection approach to this problem. While an implementation may rely solely on $\mathcal{D}''$, we will assume $\mathcal{D}''$ supports $\mathcal{D}'$ and therefore must observe invariants (RC1-4) of section 6.3. As is the case for $\mathcal{D}'$, $\mathcal{D}''$ exploits the structure of $\tau(P)$ to economic advantage.

7.2. Marking.

We offer two approaches to Ret and Act marking. The first, Mark1, directly implements the search implicit in (AR1-5). The second, Mark2, is optimal in space and time but fails in the special case of the combined presence of multi-level remote accessing and idle chain subheads.

7.2.1. Mark1. Let us assume the existence of two unused bits in each instance x: $ret(x)$ and $act(x)$, all zero between mark-sweep activations. Then Mark1, given in fig. 4, sets $ret(x) = 1$ iff $x \in \mathit{Ret}$, and $act(x) = 1$ iff $x \in \mathit{Act}$. Mark1 may be appraised as follows:

i) the *space* requirement of Mark1 may be criticized because:

  a) two bits are needed per instance, and

  b) the mutual recursion between setret and setact is bounded only by $|\mathit{Ret}|$. (A link permutation scheme along the lines of [20] might offer an iterative solution at the cost of added complexity.)

```
setret(x):   {mark x∈Ret (if not already) & pursue consequences}

if ret(x)=0 then
begin local e;
      ret(x) := 1;
      if policy(c) then  {(AR4)}
            for all e such that t(e)=x do
                  if e↑≠nil then setret(e↑);
      if x≠P and Hbit(x)=1 and x.SC.ip≠undefined and
            act(T(x))=1 then setact(x)  {(AR5)}
end


setact(x):   {mark x∈Act (if not already) & pursue consequences}

if act(x)=0 then
begin local y, e;
      act(x) := 1;  setret(x);  {(AR1)}
      if Hbit(x)=1 or x=P then  {(AR2-3)}
      begin y := D(x);
            while y≠x do
            begin setact(y);  y := D(y)
            end
      end;
      if policy(b) then  {(AR4)}
            for all e such that t(e)=x do
                  if e↑≠nil then setret(e↑);
      y := desc(x);
      while y≠nil do  {(AR5)}
      begin if ret(y)=1 and Hbit(y)=1 and y.SC.ip≠undefined then
                  setact(y);
            y := sib(y)
      end
end
```

Fig. 4. Fully general *Ret* and *Act* marking under $\mathcal{D}''$; starts with setact(P).

ii) the *time* requirement of Mark1 may be criticized because the loop in setact implementing (AR5) prevents Mark1 from running in time proportional to the number of instances marked.

7.2.2. <u>Mark2</u>. Despite these shortcomings of Mark1, no better marking algorithm has been found for the general case. If, however, we assume *either* the absence of idle chain subheads *or* the prohibition of multi-level remote accessing, another approach exists which is "ideal" in the following senses:

i) only one mark bit, *mark*(x), is used in each instance x;

ii) a purely iterative algorithm suffices, using only a fixed set of working variables, and

iii) the algorithm runs in time proportional to the number of instances marked.

In a manner similar to that used in queuechain (section 6), Mark2 (fig. 5) uses a worklist W of instances scheduled for processing. However, instances enter W by complete chains, rather than individually as in queuechain. Although such chains must retain their integrity after processing, they can temporarily be concatenated together with boundaries marked by the Hbit of each "head" instance. Initially, the OC is put on W. Then:

i) we work through W from left to right, marking the current instance x and examining its referenced instances y.

ii) each unmarked such instance y is:

   a) marked, and

   b) if y is a head instance, chain(y) is put on W.

iii) whenever the current instance x has Hbit(x) = 1, we reform chain(x) as an idle chain.

iv) when W is about to become empty, we halt.

7.3. Validating Mark2.

We will now argue the correctness of Mark2. First, three definitions.

```
Mark2:    {Single bit, iterative marking algorithm}

begin local front, rear, x, y, e;
      front := P;  rear := D(P);  x := rear;  {OC established as W}
      loop  mark(x) := 1;
            for all e such that t(e)=x do
                  if e↑≠nil then
                        if mark(e↑)=0 then
                        begin mark(e↑) := 1;
                              if Hbit(e↑)=1 then
                              begin {splice chain(e↑) at front of W}
                                    y := D(e↑);  D(e↑) := rear;
                                    D(front) := y;  front := e↑
                              end
                        end
      exit if x=front;
            if Hbit(x)=1 or x=P then
            begin {reform chain(x)}
                  y := D(x);  D(x) := rear;  D(front) := y;
                  rear := y;  x := y
            end
            else  x := D(x)
      end
end
```

Fig. 5.   Special case $\mathcal{R}et$ marking under $\mathcal{D}''$.

Wset = { x | $x \in$ W at some time during Mark2 }

Marked = { x | $\mathit{mark}(x)$ = 1 after Mark2 terminates }

$R(Q)$, where Q is a subset of S:

{ x | for some e such that $t(e) \in Q$: $e\uparrow = x \in S$ }.

The validation of Mark2 involves proving (a) its termination, and (b) that $\mathit{Marked} = \mathit{Ret}$. The proof of (a) is trivial since W expands only when new chains are marked. We prove (b) under each of the two alternative preconditions of Mark2.

7.3.1. <u>Mark2 under policy (b)</u>.

<u>Lemma 1</u>. If policy (b) holds, Wset is a subset of Act <u>union</u> T <u>intersect</u> Ret.

<u>Proof</u>. By induction on chains brought onto W. Initially, W = OC, a subset of Act by (AR2). Now consider L $\in L$ brought onto W. Necessarily, h(L) = $e\uparrow$ for some $t(e) \in$ W. By induction, $t(e) \in \mathit{Act}$ or $t(e) \in T$. Clearly $t(e) \notin T$ since $R(T) = \emptyset$ under policy (b). Hence $t(e) \in \mathit{Act}$, and by (AR4') we have $e\uparrow \in \mathit{Ret}$ and either $e\uparrow \in T$ or $e\uparrow \in \mathit{Act}$. If $e\uparrow \in T$ then $e\uparrow$ is a singleton chain; otherwise, L is a subset of Act by (AR3). In either case, the induction is complete.

<u>Lemma 2</u>. If policy (b) holds, Marked is a subset of Ret.

<u>Proof</u>. By inspection of Mark2, we observe that $\mathit{Marked} = \mathit{Wset}$ <u>union</u> $R(\mathit{Wset})$. By Lemma 1 and the definition of R we have Marked = a subset of Act <u>union</u> T <u>intersect</u> Ret <u>union</u> $R(\mathit{Act})$ <u>union</u> $R(T$ <u>intersect</u> $\mathit{Ret})$. Each of these terms is a subset of Ret by either (AR1-4') or policy (b).

<u>Lemma 3</u>. If policy (b) holds, Act is a subset of Wset.

<u>Proof</u>. By induction on applications of (AR1-AR4') that bring instances into Act. Clearly OC is a subset of Act. Assume that all y currently in Act are in Wset. If x joins Act by (AR3), necessarily $x \neq$ h(chain(x)), already in Act. Hence h(chain(x)) $\in \mathit{Wset}$, and so is x.

Otherwise, x must join Act by (AR4'), with some $t(e) \in \mathit{Act}$ already and $e\uparrow = x$, $x \in H$. By induction $t(e) \in \mathit{Wset}$, so chain(x) will enter W when $t(e)$ is current, if not earlier.

Theorem 8. If policy (b) holds, $\mathit{Ret} = \mathit{Marked}$.

Proof. By Lemma 2, Marked is a subset of Ret. It remains to show Ret is a subset of Marked. By (AR1-4'), $\mathit{Ret} = \mathit{Act}$ union $R(\mathit{Act})$. By Lemma 3, Act is a subset of Wset, which is a subset of Marked. Similarly, $R(\mathit{Act})$ is a subset of $R(\mathit{Wset})$, which is a subset of Marked.

7.3.2. Mark2 under subhead prohibition.

Lemma 4. $x \in \mathit{Wset}$ implies $x = P$ or $T(x) \in \mathit{Wset}$.

Proof. Follows readily by CRF and induction on instances entering W.

Lemma 5. If subheads are prohibited, $\mathit{Marked} = \mathit{Wset}$.

Proof. By inspection of Mark2, Wset is a subset of Marked. We now show that Marked is a subset of Wset. Consider $x \in \mathit{Marked} - \mathit{Wset}$. Necessarily for some $t(e) \in \mathit{Wset}$, $e\uparrow = x \notin H$. By CRF, $t(e) \overset{*}{\Rightarrow} T(x)$; by Lemma 4, $T(x) \in \mathit{Wset}$. If $T(x) \in$ chain(x), $x \in \mathit{Wset}$, a contradiction. Otherwise x is a subhead, also a contradiction.

Theorem 9. If subheads are prohibited, $\mathit{Marked} = \mathit{Ret}$.

Proof. By Lemma 5, we may prove $\mathit{Wset} = \mathit{Ret}$. Clearly from Mark2, Wset is a subset of Ret. To show Ret is a subset of Wset, we argue inductively on the formation of Ret. Initially Ret = OC, in Wset by initialization of W. Consider now $x \notin$ OC entering Ret, under the inductive hypothesis that Ret thus far is a subset of Wset. If x enters Ret by (AR4), $t(e) \in \mathit{Ret}$ already and $e\uparrow = x$. Hence $t(e) \in \mathit{Wset}$, implying $x \in \mathit{Marked}$, implying $x \in \mathit{Wset}$ by Lemma 5. Otherwise, x must enter Ret by (AR1), implying by our standard construction order that already h(chain(x)) $\in \mathit{Act}$ and h(chain(x)) $\in \mathit{Ret}$. By induction, h(chain(x)) $\in \mathit{Wset}$, implying $x \in \mathit{Wset}$.

7.4. Sweeping.

The rules for instance deletion and termination given in section 5.3 provide that at any time we may:

$$\begin{array}{l} \textit{delete}\quad S - \mathit{Ret}, \text{ and} \\ \textit{terminate}\ \mathit{Ret} - \mathit{Act} - T. \end{array}$$

We now examine efficient means for accomplishing this storage regeneration given simply knowledge of Ret and T. Ret may have been determined either by Mark1 ($ret(x) = 1$) or Mark2 ($mark(x) = 1$); notationally, we assume the latter. T is, of course, characterized by x.SC.ip = *undefined.*

7.4.1. The set N. Let the set of instances needing termination (i.e. $\mathit{Ret} - \mathit{Act} - T$) be called N. We now prove an important property of N permitting its recognition without explicit knowledge of Act.

Lemma 6. $x \in N$ implies $x \neq P$ and ($T(x) \in N$ or x is a subhead).

Proof. Clearly $x \in N$ implies $x \notin \mathit{Act}$, so $x \neq P$. Hence $T(x)$ exists and is in $\mathit{Ret} - T$ by (AR7) and (RC4b). It remains to show $T(x) \notin \mathit{Act}$ or x is a subhead. If $x \in H$, $T(x) \notin \mathit{Act}$ by (AR5) since $x \notin \mathit{Act}$. Now suppose $x \notin H$; if x is a subhead, we are done; otherwise $T(x) \in$ chain(x) and again $T(x) \notin \mathit{Act}$ since chains are each either entirely in Act or entirely disjoint from Act.

7.4.2. Sweeping with no subheads. Lemma 6 leads directly to the following useful fact:

Theorem 10. If idle chain subheads are prohibited, N is empty.

Given Theorem 10, it is worthwhile to implement sweeping under subhead prohibition as a special case. Sweep1, given in fig. 5, provides an efficient approach that is scope-based, i.e. involves traversal of $\tau(P)$. Since we are assuming $\mathcal{D}''$ is being implemented in support of $\mathcal{D}'$, $\mathcal{D}''$ must decrement *ext* counts when destroying references to members of Ret. However, during the operation of $\mathcal{D}''$ we do not wish $\mathcal{D}'$ to be triggered, so we assume schedule calls within remref are bypassed. As in deletedesc, we are cautious not to delete any instance that may later have its *ext* count decremented; hence two passes through each descendant list are used. This would not be necessary

```
sweep1(x):    {fast sweep algorithm, applicable if no subheads;
                    given  x ε Ret, delete y∈τ(x) with mark(y)=0;
                    clear all mark bits except that of x;
                    all z referenceable from within τ(x) still exist}


begin local w, y, z;
      {recursion and ref clearing pass over desc. list of x}
      y := desc(x);
      while y≠nil do
      begin if mark(y)=0 then    {y ∉ Ret}
                  deletedesc(y, y)
            else sweep1(y);
            y := sib(y)
      end;
      {deletion and mark clearing pass over desc. list of x}
      y := desc(x);  z := x;  {z trails y on desc. list of x}
      while y≠nil do
      begin w := sib(y);
            if mark(y)=1 then    {y ε Ret}
            begin mark(y) := 0;  z := y
            end else   {y ∉ Ret}
            begin spliceout(y, z);      {remove y from list, using z}
                  destroy(y)
            end;
            y := w
      end
end
```

Fig. 5. Special case sweeping under $\mathcal{D}''$; starts with sweep1(P).

if $\mathcal{D}''$ were being implemented alone.

The performance of sweep1 is excellent, since its time requirement is linear with $|S|$ (despite the two-pass approach), and its space requirement is bounded by the maximum static depth of the program.

7.4.2. Sweeping with subheads. If idle chain subheads are possibly present, we must see to the termination of N instances. However, this may still be done in a scope-based manner, as shown by the following theorem:

Theorem 11. If $T(x) \in Ret - N$, then $x \in N$ iff (i) $x \in Ret$, (ii) x is a subhead, and (iii) h(chain(x)) $\notin Ret$.

Proof. If $T(x) \in Ret - N$, $T(x) \in Act$ or $T(x) \in T$. If $T(x) \in T$, $x \notin N$ and $x \in T$ implying x is not a subhead. We now assume $T(x) \in Act$, and show both sides of our equivalence.

Suppose $x \in N$. Then (i) $x \in Ret$, and (ii) x is a subhead by Lemma 6. Since $x \notin T$, h(chain(x)) $\notin T$; if h(chain(x)) $\in Ret$ then it would be in Act by (AR5), as would x by (AR3). Thus (iii) is established.

Now suppose (i) - (iii) hold. By (i) $x \in Ret$; by (ii) $x \notin T$ since T members are singleton chains. By (iii) h(chain(x)) $\notin Act$, so $x \notin Act$. Hence $x \in N$.

By Theorem 11, we can recognize "top-level" N instances by searching for marked subheads on chains with unmarked heads. If these are terminated as encountered, lower level N instances may be detected by the anomalous condition of $T(x) \in T$ and $x \notin T$.

The search for such top-level N instances can be incorporated into sweep1 as an extra phase performed on the descendants of x when $x \in Ret - N - T$ (see the revised algorithm sweep2 in fig. 6). Note that this search cannot be combined into the existing phases of sweep1, since we cannot recognize a subhead y after chain(y) has begun being broken up.

To estimate the speed of sweep2, let us assume for the moment that *disp* computations can be done in fixed time (e.g. each instance contains its own display vector). Then despite the two nested loops new to sweep2, time linear with $|S|$ remains. This is because within the overall computation of

```
sweep2(x):    {same conditions as SWEEP1, except that idle chain
                    subheads are permitted}

begin local w, y, z;
      if x.SC.ip≠undefined then
            if x≠P and T(x).SC.ip=undefined then
                  term(x)   {rippled termination}
            else  {look for w=>x needing termination}
            begin y := desc(x);
                  while y≠nil do
                  begin if Hbit(y)=1 and mark(y)=0 then
                        begin {terminate marked subheads of chain(y)}
                              z := D(y);
                              loop  w := disp(z, stat(y));
                              exit if w=y; {w is next subhead (or y)}
                                    z := D(w);
                                    if mark(w)=1 then
                                          term(w)
                              end
                        end;
                        y := sib(y)
                  end
            end;
      .
      .     {remainder is code of SWEEP1, but with recursive call
      .           replaced with call on SWEEP2}
      .
      .
end
```

Fig. 6. Fully general sweeping under D''; starts with sweep2(P).

sweep2(P), the outer loop is executed at most once for each $y \in S$, and the inner loop is executed at most once for each head and subhead in S. If *disp* requires T-link searching we must include this time factor, bounded by the maximum static depth of the program. The space for sweep2 is the same as that for sweep1.

8. SUMMARY AND FUTURE WORK.

8.1. Summary.

Using the control description approach of Wang and Dahl as a base, we have explored the effects of static program structure on BSCRs in a compilation-oriented setting. Conditions on instance reference assignment (CRF) and individual control actions (CI, CC, CD, CR) were defined and implemented. The retention requirements of our class of BSCR languages were postulated, and used as the formal basis for the development of two compatible deletion strategies. The first, $\mathcal{D}'$, uses scope-based reference counting to incrementally detect instance deletion and termination opportunities. The second, $\mathcal{D}''$, does mark-sweep garbage collection to reclaim isolated cyclic structures overlooked by $\mathcal{D}'$.

The results obtained here suggest that detach operations in BSCR languages should refer to program units statically surrounding the currently operating instance. If this design choice is made, then our implementation approaches for both condition testing and instance deletion are highly efficient in both space and time.

8.2. Future work.

One omission in this work is the consideration of reference-valued parameters. If, however, these are either excluded or made to obey CRF, our results stand. It may well be fruitful to explore design choices intermediate between this extreme and that of unrestrained reference passage via parameters. One approach might be the explicit declaration of which references can be exported or imported; such constructs are now being advocated as a means toward better program modularization.

Another area for continuing work concerns the adaptation of these results to execution environments involving true concurrency [13]. Some preliminary studies have indicated that process reference counting is preferable to garbage collection, in light of the latter's apparently greater need for centralized control [18]. How scoping issues could be exploited in this setting remains unknown.

Finally, additional work is merited on the notion of control discipline *necessity* given particular implementation strategies. This line of inquiry reverses that explored in this paper, in which sample disciplines were shown to be *sufficient* to safeguard the correctness of an implementation strategy [15]. With insights in this direction, important questions such as the class of BSCR programs for which $\mathcal{D}''$ is superfluous might be approached.

REFERENCES

1. Baker, Henry G., Jr., and Carl Hewitt, "The incremental garbage collection of processes," Proc. Symp. AI & Prog. Lang., SIGPLAN Notices 12,8 (Aug. 1977) 55-59.

2. Berry, D.M., L.M. Chirica, J.B. Johnston, D.F. Martin, and A. Sorkin, "Time required for reference count management in retention block-structured languages," Int'l. J. Comp. & Inf. Sci., Part 1: 7,1 (1978) 11-64; Part 2: 7,2 (1978) 91-119.

3. Berry, D.M., and A. Sorkin, "Time required for garbage collection in retention block-structured languages," Int'l. J. Comp. & Inf. Sci. 7,4 (1978) 361-404.

4. Bobrow, Daniel G., and Ben Wegbreit, "A model and stack implementation of multiple environments," Comm. ACM 16,10 (Oct. 1973) 591-603.

5. Clint, M., "Program proving: coroutines," Acta Inf. 2 (1973) 50-63.

6. Conway, M.E., "Design of a separable transition-matrix compiler," Comm. ACM 6,7 (July 1963) 396-408.

7. Dahl, Ole-Johan, "An approach to correctness proofs of semicoroutines," in Programming Methodology, Springer-Verlag (1978) 116-129.

8. Dahl, Ole-Johan, B. Myhrhaug, and K. Nygaard, "SIMULA-67: common base language," Norwegian Computing Center Publ. S-2, Oslo (1968).

9. Friedman, Daniel P., and David S. Wise, "CONS should not evaluate its arguments," Tech. Rpt. 44 (Nov. 1975), Dept. of Comp. Sci., Indiana Univ.

10. Griswold, Ralph E., and David R. Hanson, "Language facilities for programmable backtracking," Proc. Symp. AI & Prog. Lang., SIGPLAN Notices 12,8 (Aug. 1977) 94-99.

11. Hanson, David R., and Ralph E. Griswold, "The SL5 procedure mechanism," Comm. ACM 21,5 (May 1978) 392-400.

12. Ingalls, D., "The Smalltalk-76 programming system," Proc. 5th Symp. Princ. Prog. Lang., Tucson (1978) 9-16.

13. Kahn, Gilles, and David MacQueen, "Coroutines and networks of parallel processes," IRIA/LABORIA Research Rpt. 202 (Nov. 1976).

14. Krieg, Bernd, "A class of recursive coroutines," IFIP Congress 74, North-Holland (1974), Software Booklet 408-412.

15. Lindstrom, Gary, and Mary Lou Soffa, "Control discipline necessity: making the language as general as the implementation," forthcoming tech. rpt., Dept. of Comp. Sci., Univ. of Utah.

16. McDermott, Drew, and Gerald Jay Sussman, "The CONNIVER reference manual," MIT AI Lab Memo 259 (May 1972).

17. McIlroy, M. Douglas, "Coroutines: semantics in search of a syntax," unpublished manuscript (1968).

18. Nori, Anil Kumar, "A storage reclamation scheme for Applicative Multi-Processing System AMPS," M.S. thesis (Sept. 1979), Dept. of Comp. Sci., Univ. of Utah.

19. Randell, B., and L.J. Russell, _Algol 60 Implementation_, Academic Press (1964).

20. Schorr, H., and W.M. Waite, "An efficient machine-independent procedure for garbage collection in various list structures," _Comm. ACM_ 8,10 (Aug. 1967) 501-506.

21. van Wijngaarden, A., B.J. Mailloux, J.E.L. Peck, C.H.A. Koster, M. Sintzoff, C.H. Lindsey, L.G.L.T. Meertens, and R.G. Fisker, "Revised report on the algorithmic language Algol 68," _Algol Bull._ 36 (March 1974).

22. Vanek, Leonard I., "Hierarchical coroutines: a mechanism for improved program structure," Tech. Rpt. 99 (Feb. 1979), Comp. Sys. Research Group, Univ. of Toronto.

23. Wang, Arne, and Ole-Johan Dahl, "Coroutine sequencing in a block-structured environment," _BIT_ 11 (1971) 425-449.