# CCAL: An Interpreted Language for Experimentation in Concurrent Control*

Phil Kearns
Department of Computer Science
The College of William and Mary
Williamsburg, VA 23185

Chris Cipriani
Tartan Laboratories
Pittsburgh, PA 15213

Mitzi Freeman
Department of Computer Science
University of Pittsburgh
Pittsburgh, PA 15260

## Abstract

Concurrent Control Abstraction Language, CCAL, is an interpreted language which provides no particular control regime to the user. CCAL instead supports five primitive operations which manipulate an abstract model of inter-procedural control. This model is intrinsically concurrent, and the user is allowed to construct high-level concurrent control operations from the primitives (hence, control abstraction). The primary use of CCAL is as a vehicle by which rapid prototyping of application specific control forms may be done and as a tool for the construction and evaluation of novel control forms, especially control forms for highly concurrent and distributed systems. The CCAL interpreter is implemented as a distributed program on a network of Vaxen and Sun-3 workstations under 4.2bsd and 4.3bsd Unix[1]. CCAL programs appear as multi-process programs in a shared memory system. Both true and apparent concurrency are possible. This paper describes the control abstraction facilities offered by the CCAL interpreter, its use, and implementation strategies in the distributed environment.

## 1. Introduction

The text of a standard procedural high level language program consists of a number of program *components* (procedures, functions, subroutines, coroutines, tasks, etc.). During execution, the locus/loci of control will migrate between *instances* of those components in accordance with the inter-component *control regime* provided by the language. Common examples of control regimes include:

- non-recursive procedures in Fortran-77;
- recursive procedures in Pascal or Algol-60;
- recursive procedures and coroutines in Simula-67;
- recursive procedures and concurrent tasks (with synchronous rendezvous as the inter-task communication mechanism) in Ada[2].

It is commonly accepted that the control regime supported by the language is a prime factor in the ease of coding certain classes of applications in the language. An inappropriate control regime will be reflected in an inappropriate mind set on the part of the programmer which may lead to code which is intrinsically obscure, error-prone, and difficult to maintain. We postulate that this issue becomes even more critical in a concurrent program, where timing and synchronization concerns compound the problem. In a distributed program, all of the problems of concurrency are present, but additional complexity is introduced due to the possible loss or corruption of messages and the failure of processors.

Investigating new control regimes is a difficult and time-consuming problem – it is generally done in the context of the design and implementation of a new programming language. As such, we contend that many of the problems in the construction of large programs may be due to the "squeezing" of the application into an inappropriate control regime. CCAL is an interpreted high-level language in which control regimes, including concurrent

*This work supported in part by NSF Grant DCR81-19341.
[1] Unix is a registered trademark of AT&T Bell Laboratories.



[2] Ada is a registered trademark of the U.S. Government (Ada Joint Program Office).

control regimes, may be constructed within the confines of the language itself (much like the use of data abstraction to create new data types). Although there have been several previous works on control abstraction, they have either been restricted to purely sequential control[THLZ77,Tur84,Tur83,Lew79], or have not been intended as a practical mechanism for prototyping control regimes[Fis70].

In order to describe the control operations provided to the user by CCAL, we first present the formal operational control model upon which it is based. This model is a generalization of the Wang and Dahl model[WD71] for coroutine semantics. In addition to serving as the semantic basis for CCAL control, the model has been used in several formal studies of storage management for high-level languages[KS83,KQ84,Qua86]. Thus it is possible to deal formally with control regimes described and implemented through this model.

# 2. The Control Model

## 2.1. The Control State

The control state of a program in execution is expressed as a tuple,

$$(\Pi, \Sigma, D, status, processor).$$

Here,

$$\Pi = \{\pi_1, \pi_2, \ldots, \pi_n\}$$

is the set of physical processors available to the program. We currently assume that this set is fixed throughout the program's execution.

$\Sigma$ is the set of program component instances – an activation record is the typical implementation of an instance. $\Sigma$ will expand and shrink as the program executes, creating and deleting instances. We find it convenient to denote $\Sigma = T \cup P$, where $T \cap P = \emptyset$. Elements of $T$ are considered to be *task instances* capable of being scheduled for execution on a processor and of supporting a separate logical thread of control. Elements of $P$ are *non-task instances* – coroutine instances and procedure instances are the obvious examples. In order for an element of $P$ to execute, it must be invoked (perhaps indirectly) by a task. The mechanism for that invocation will be described concretely below.

The *status* function,

$$status: \Sigma \rightarrow \{ready, nonready\},$$

denotes the execution status of an instance (it is most appropriate for tasks). A *ready* task is logically executing; it will actually execute without any action of the program. A *nonready* task cannot execute until some action in another task of the program makes it eligible.

The *processor* function,

$$processor: \Sigma \rightarrow \Pi,$$

denotes the processor on which a given instance currently resides.

The dynamic enclosure function,

$$D: (\Sigma \cup \Pi) \rightarrow (\Sigma \cup \Pi),$$

imparts structure to the control state. As a notational convenience, we define the relation

$$\Rightarrow \subseteq (\Sigma \cup \Pi) \times (\Sigma \cup \Pi),$$

where for $x, y \in (\Sigma \cup \Pi)$, $x \Rightarrow y$ iff $D(x) = y$. $\stackrel{*}{\Rightarrow}$ denotes the transitive and reflexive closure of $\Rightarrow$. Note that $\mathcal{G} = ((\Sigma \cup \Pi), \Rightarrow)$ is a graph which is an essential component of the control state. This graph allows us to determine the current sites of execution under the following convention:

**Execution Condition**: *A locus of control is in instance $x \in \Sigma$ (or, equivalently, instructions in the component corresponding to instance $x$ are being executed) iff for some $\pi \in \Pi$, $D(\pi) = x$. Further, $x$ is actually in execution on the processor processor($x$).*

The execution condition, by itself, simply states that a given instance may be in execution if it is the "dynamic parent" of a processor. How it becomes the dynamic parent of the processor is a function of how various *primitive control operations* are applied against the control state. It should be noted that the actual site of execution need not be the processor of which $x$ is the dynamic parent – it is the processor at which $x$ is allocated.

The initial control state is

$$\sigma_0 = (\Pi, \{main\}, D_0, status_0, processor_0,)$$

where *main* denotes the instance corresponding to the main program (a task, by convention). Further,

- $D_0(main) = \pi \in \Pi$,
- $D_0(\pi) = main$,
- $status_0(main) = ready$, and
- $processor_0(main) = \pi$.

In terms of what we know about the control model at this point, an instance of the main component is executing on processor $\pi$ (this follows because $D(\pi) = main$). The primitive control operations, alluded to above, may be viewed as functions which map a given control state into another.

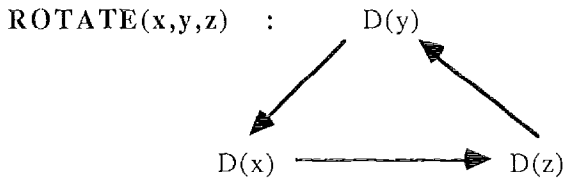
## 2.2. Primitive Control Operations

In order to define the primitive control operations, we first define three sets derived from the control state and an operation which alters the structure of the control state. The three sets are:

For $x \in \Sigma$, $task(x) = \{y \mid y \in T \wedge x \stackrel{*}{\Rightarrow} y\}$;
$H = \{x \mid (x \in \Pi)$
$\vee (x \in P \wedge status(x) \neq ready)$
$\vee (x \in T \wedge \nexists \pi \in \Pi : x \stackrel{*}{\Rightarrow} \pi)\}$;
For $x \in \Sigma$, $head(x) = \{y \mid y \in H \wedge x \stackrel{*}{\Rightarrow} y\}$.

Clearly, $task(x)$ identifies any task instances reachable from $x$ by $\Rightarrow$ links. $head(x)$ identifies the "primary" instances reachable from $x$. If $x$ or one of its dynamic children is in execution, then an element of $\Pi$ must be in $head(x)$; otherwise, the head set of $x$ will consist of *nonready* non-task instances or tasks which cannot reach a processor via $\Rightarrow$ links.

$\mathcal{G}$, the graph of instances linked by dynamic enclosure, is altered primarily by the application of a three way swap operation. The operation **ROTATE**$(x, y, z)$ circularly exchanges the values of $D(x)$, $D(y)$, and $D(z)$ as shown below:




If any single argument of **ROTATE** is undefined ($\emptyset$), then the exchange of values is short-circuited around the undefined argument. For example, **ROTATE**$(x, \emptyset, z)$ simply swaps the values of $D(x)$ and $D(z)$. If two or three arguments of **ROTATE** are undefined, the **ROTATE** is effectively a no-op.

There are four primitive control operations which may be executed by an instance to alter the control state. *All of these primitives must be viewed as atomic.* In each case, let **invoker** denote the executing instance which actually executes the corresponding primitive control operation.

**create**$(X, \pi)$
For some program component $X$ and $\pi \in \Pi$,

1. $x$, a new instance of $X$, is created;
2. if $X$ is a task, $T := T \cup \{x\}$;
3. if $X$ is not a task, $P := P \cup \{x\}$;
4. $status(x) := nonready$;
5. $D(x) := x$;
6. $processor(x) := \pi$.

**exchange**$(x, y)$
For some $x, y \in \Sigma$,

1. **ROTATE**$(head($**invoker**$), x, y)$;
2. $status(x) := ready$;
3. $status(y) := nonready$.

**activate**$(x)$
For some $x \in \Sigma$,

$status(x) := ready$;

**terminate**$(x)$
For some $x \in \Sigma$,

1. **ROTATE**$(head(x), x, \emptyset)$;
2. $\forall y \in \Sigma: y \stackrel{*}{\Rightarrow} x$: $\Sigma := \Sigma - \{y\}$;

It should be noted that the use of these operations may, in general, lead to bizarre and potentially meaningless control states. We therefore define *application conditions* for each operation. These conditions are specified in Table 1.

| Operation | Condition |
|---|---|
| **create**$(X)$ | **true** |
| **exchange**$(x, y)$ | $x \notin T \wedge x \in H \wedge$ **invoker** $\stackrel{*}{\Rightarrow} y$ |
| **activate**$(x)$ | $x \in T$ |
| **terminate**$(x)$ | **invoker** $\stackrel{*}{\Rightarrow} x$ |

Table 1: Application Conditions for Control Primitives

## 2.3. The Scheduler

The one remaining entity in the model is the *scheduler*. The scheduler is the sole agent by which a task may be linked onto a processor; a user program may **exchange** or **terminate** a task off a processor. Thus, the scheduler determines which instances are physically in execution. The scheduler implements the following primitive control operation:

**schedule**$(\pi, t)$
For some $\pi \in \Pi$ and $t \in T$,

1. $status(t) := ready$;
2. **ROTATE**$(task(\pi), \pi, t)$.

The sole application condition for **schedule** is that $status(t) = ready$. The precise mechanism by which the scheduler is invoked is irrelevant here. The CCAL scheduler is a simple round-robin timeslice scheduler – *ready* instances on a given processor fairly share access to that processor via the timeslicing.

## 2.4. Control State Invariants

Given the scheduler and the user-invoked primitive control operations, and given the initial control state, one may readily construct inductive arguments to produce the following control state invariants.

**C1.** $\Rightarrow$ is a cyclic permutation of $\Sigma \cup \Pi$.

**C2.** for $s, t \in T$: $s \stackrel{*}{\Rightarrow} t \supset s = t$.

**C3.** for $\pi_1, \pi_2 \in \Pi$: $\pi_1 \stackrel{*}{\Rightarrow} \pi_2 \supset \pi_1 = \pi_2$.

**C4.** $\forall x \in \Sigma$: $\mid head(x) \mid = 1$.

**C5.** $\forall x \in \Sigma: \pi \in \Pi: x \Rightarrow \pi \supset x \in T \wedge status(x) = ready$.

In other words, $\mathcal{G}$ consists of a number of disjoint circular chains of instances. A chain may contain any number of non-task instances, but at most a single task instance, and at most a single processor. If the chain is headed by a processor it is termed an *operating chain*, and the processor must be the dynamic parent of a *ready* task instance. The dynamic parent of the processor is the active instance on that operating chain. If the chain is headed by a task instance (*ready* or *nonready*), then the chain represents a suspended task (*nonready*) or a logically active task (*ready*) which is waiting to be scheduled for execution. If a chain is headed by a non-task instance, that instance must be *nonready*; the chain would most intuitively correspond to a detached coroutine.

In this structured context, one may see that a **create** merely produces an idle chain consisting of the newly created instance. **exchange**$(x, y)$ serves to detach the tail of the operating chain at instance $y$, thereby forming an idle chain headed by $y$. The **exchange** also grafts some idle chain headed by $x$ onto the end of the operating chain. **activate** marks some task instance as *ready*. **terminate**$(x)$ detaches that portion of the operating chain rooted on $x$ and then deletes those detached instances from $\Sigma$. **schedule** links chains headed by ready tasks onto processors, and (by the execution condition) brings instances into execution.

Figure 1 shows a program in execution under this model. Task instance $T_1$, having invoked non-task instance $P_1$, executes on processor $\pi_1$. Task $T_2$ has called non-task instance $P_2$ (most likely through an **exchange**$(P_2, \emptyset)$), and $P_2$ is actually in execution on $\pi_2$. Processor $\pi_3$ is idle. Task $T_3$ has been suspended after having invoked non-task $P_3$. The idle chain headed by $P_4$ most likely is a detached coroutine chain.

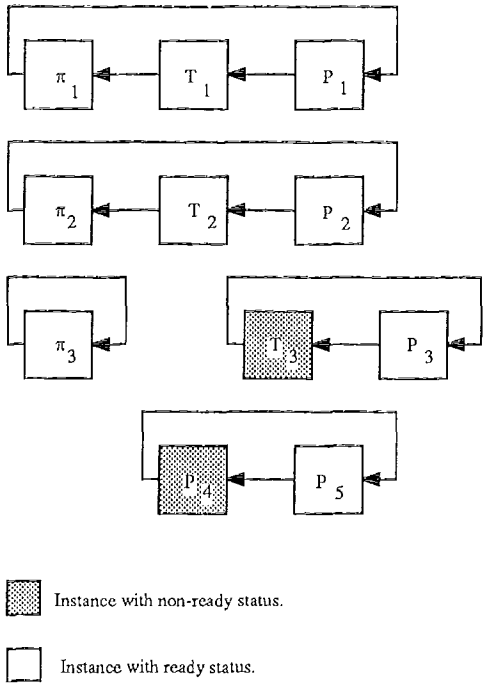



Figure 1: An Example of $\mathcal{G}$

## 2.5. Aggregate Control Operations

Useful high level control forms may be readily constructed from combinations of the primitive control operations. A traditional procedure call would be implemented as:

$$\begin{aligned} \textbf{Call}(X) \equiv [ & \\ & x := \textbf{create}(X); \\ & \textbf{exchange}(x, \emptyset);] \end{aligned}$$

Here we have taken the liberty of using **create** as a function which returns the value of the newly created instance of $X$. $x$ is a variable typed as an instance (cf., environment-valued variables in SL5[HG78]). The brackets, [...], denote an aggregate operation. The code between the brackets may be best viewed as an **atomic transaction** applied against the control state. A sequence of candidate control states is produced as individual operations within the brackets are executed; only when the closing bracket is successfully reached will the final candidate control state be embedded as the new control state. Failing aggregate operations do not impact upon the control state.

Procedure return is then:

**Return** $\equiv$ [
**terminate**(**self**);]

Here **self** is an instance constant; its value is that of the instance corresponding to the component in which it is used. We have not dealt with parameter and functional value transmission in the above. We consider it not properly a control activity; if it were necessary in a particular modeling context, however, a mechanism similar to that in SL5 could be embedded in the basic model. Parameter transmission is explicitly handled in CCAL.

The basic operations for a coroutine control regime may be constructed using the same primitive control operations. They are a straightforward generalization of the procedural regime. A traditional coroutine detach would employ an **exchange**($\emptyset$,?) operation in some fashion in order to produce an idle chain of non-task instances headed by the instance specified by the second argument; a conventional resume would employ an **exchange**(?,$\emptyset$) in order to graft the idle chain headed by the instance specified as the first argument onto the operating chain behind the invoking instance.

A more substantial control paradigm, one which is intrinsically concurrent, is the Ada rendezvous. If an Ada task elaborates an entry $e$, the task must (effectively) allocate a queue of instances to serve as the waiting line for callers of that entry since Ada requires FCFS entry call service.

**Elaborate_entry**($e$) $\equiv$ [
$q_e$ := *queue*;]

If a task now attempts to invoke entry $e$, assuming $e$ to be elaborated in an Ada task $t$, the invocation would be implemented as:

**Entry_call**($t, e$) $\equiv$ [
*enqueue*(**self**,$t.q_e$);
**activate**($t$);
**exchange**($\emptyset$, *task*(**self**);]

The corresponding "**accept** e" in the code of $t$ would be:

**Accept**($e$) $\equiv$ [
**while** *empty*(**self**.$q_e$)
**do exchange**($\emptyset$,**self**);]

The end of the accept is:

**End_accept**($e$) $\equiv$ [
$x$ := *behead*($q_e$);
**activate**($x$);]

The *enqueue*, *empty*, and *behead* functions (applied to a queue of instances) have the obvious semantics. The *task*($x$) function returns the identity of the task at the head of the chain which includes $x \in \Sigma$. Note the liberal use of auxiliary data structures in order to construct this particular control regime.

# 3. CCAL Constructs

CCAL is a faithful implementation of the above control model. The current implementation of CCAL provides limited data types and relatively simple intra-component control (Pascal-like loops and conditionals). A CCAL program is a (non-nested) sequence of program components, template declarations, and control operation definitions.

A **program component** is the only subprogram unit. There is no static distinction made between tasks and non-tasks. One component must be named "main" – this component is created as a task on one processor of the network, and execution begins at its first instruction. Components may have associated parameters, passed by value or by copy-restore (**var** parameters).

A **template declaration** declares data structures which will be included in every instance. The effect is to customize the activation record of instances in such a way as to support the newly created control regime. Multiple template declarations are effectively merged into one. There are three default user-accessible data items in every activation record:

*type* takes a value from the set {**task**,**proc**}. If the instance is schedulable, this field must have value **task**.

*dlink* denotes the dynamic parent of the instance.

*slink* denotes the "static parent" of the instance. This is used to define scoping relationships for resolving non-local data references.

**Control operations** are basically macro definitions, similar in form to the examples of the previous section. The text of the control operation is expanded in-line at the point of invocation in any component. Data declarations which may be created as part of that expansion are percolated upwards to the declarative part of the enclosing component. The code necessary to make the operation atomic, the CCAL equivalent of the brackets in the previous section, is inserted before and after the text of the operation. We accepted the syntactic ugliness of the macro approach, as opposed to an approach in which the operation is invoked as a subroutine (done in [Lew79] and [THLZ77], suggested in [Hil83]), for reasons of efficiency.

There are only three data types currently supported by CCAL: integers, strings, and instances. Strings may be used only in output statements. Instances refer to dynamic component instances (elements of $\Sigma$ in the above model). *dlink* and *slink* fields in an activation record are instance variables. The instance-valued constants **nil** and **self** refer to no instance and the instance in which **self** is used, respectively. Reference to data in an instance is made using genitive (dot) notation — if $i$ is an instance variable which denotes an instance containing integer *val*, the expression *i.val* is evaluated as that integer value. Arrays of integers and instances are supported.

Scoping rules are non-standard. Data may be declared in program components or at a global level (the same static level as the program components themselves). Static scoping relationships between components may be specified by means of *slink* paths. If instance $x$ is specified to be the static parent of instance $y$, then $y.slink = x$, and the scoping effect is as if component $Y$ had been nested inside of component $X$ in a statically scoped block structured language. The ability to dynamically alter an instance's *slink* permits the construction of arbitrary (and potentially dangerous) scoping regimes. A reference to variable $x$, made during the execution of instance $i$, is resolved according to the following convention:

1. If $x$ is declared in $I$, resolve the reference locally.
2. If $x$ is not local, follow the *slink* chain starting at $i$. If some instance along that chain declares $x$, resolve the reference at the first such instance.
3. If $x$ cannot be resolved along the *slink* chain, resolve it globally.
4. If all three steps above fail, a data reference error is raised.

The control primitives of the previous section are supported directly. Their use is restricted to be within a control operation definition. CCAL's instance creation operation deserves further explanation. Instance creation is achieved through a construct of the form:

create(<component>,<type>,<site>)
{with <expression_list> }

< *component* > must be the name of the program component of which a new instance is to be created. < *type* > must be either **task** or **proc**; **task** results in a schedulable instance. < *site* > must be the name of an Internet host on which the CCAL interpreter is running. The instance will be allocated at that site. **create** returns a value which denotes the newly created instance. The **with** clause assigns values to parameters. The **with** may also be stand-alone, permitting parameter assignment to take place at times other than instance creation.

The following example, Figure 2., illustrates the use of CCAL to construct a novel control form for use in a distributed program. The new form is that of a triply redundant (remote) procedure call similar to that suggested in [Coo84]. Upon a *triplecall*, the main component will issue three concurrent calls to the same function on three different processors; only the value returned by the first call to complete will be used. This control regime would conceivably be useful in an application which requires resilience to processor failure (the function call will survive the loss of two out of three processors) or soft real-time response (presumably, one of the three processors would be lightly loaded). We do not address efficiency, especially the issue of the extermination of live function replicates upon the first completion. The reader must recall the atomicity property of constructed control operations in order to understand the operation of this regime.

# 4. Implementation Issues

The CCAL interpreter runs as a daemon process on each site of the network. In a sense, the daemons should be thought of as the constituents of $\Pi$. One site, the "master", configures the rest of the system and establishes the appropriate communication links. The main component always begins execution at the master. The other sites ("slaves") are, with only a few exceptions, identical copies of the master. Each interpreter copy is involved in three activities:

- traversing the code tree which is the interpretable representation of the currently active instance at that site;
- scheduling executable instances which reside locally; and
- communicating with other daemons in order to maintain the control state.

Traversal of the code tree is straightforward. The flexibility of CCAL does, however, impose certain implementation difficulties. Storage management is non-trivial. Ultimately, a general retentive strategy in which absence of an access path results in deallocation of instances will be implemented. The current prototype implementation simply never deallocates storage – this was felt to be acceptable for the experimental nature of the system. The general scoping constructs require the maintenance of an association

```
#template
caller : instance;      -- Denotes the instance which
                           issues triple call.
done : integer;         - Set if triple caller has
                           received reply.
retval : integer;       -- Return value to caller from
                           first to complete.
#end

#operation triplecall(comp,loc1,loc2,loc3);
id : instance;
{     self.done := 0;                          -- Clear reporting
                                               - flag.
      id := create(comp,task,loc1);            - Create replicate.
      id.caller := self;                       - Note its master.
      activate(id);                            - Start it.
      id := create(comp,task,loc2);
      id.caller := self;
      activate(id);
      id := create(comp,task,loc3);
      id.caller := self;
      activate(id);
      exchange(∅,task(self));                  - Await the
                                               - first reply.}

#operation triplereturn(value);
- If first done, embed result in caller.
{     if caller.done = 0 then {   caller.done := 1;
                                  caller.retval := value;
                                  activate(caller);}
      terminate(self);}

component triplefunction;
-This is the routine which is triply called.
{     slink := dlink;
     ... Compute the value (complex code here) ...
     triplereturn(value);}

component main;
-Main program; will triply call triplefunction
-at nodes s1, s2, and s3.
{       ... Set up the call, etc. ...
        triplecall(triplefunction,"s1","s2","s3");
        ... Use the integer retval. ... }
```

Figure 2: Example – Triply Redundant Remote Procedure Calls

list binding symbolic identifiers to storage locations for each instance. The resolution of a non-local reference results in the explicit traversal of *slink*s and a search for an associated identifier at each level. We see no convenient mechanism for overcoming this brute-force approach without sacrificing generality in scoping.

Local scheduling is deceptively subtle in the distributed context. The difficulty lies in the fact that a non-task instance $x$ at site $S$ may or may not be schedulable depending upon conditions at sites other than $S$. For example, Figure 3 shows the case in which the execution status of $P_1$ depends upon the state of site $R$. $P_1$'s execution will consume cycles on processor $\pi_2$ at site $S$ since we do not wish to absorb the cost of remote execution. In fact, $\pi_1$ is idle, and $\pi_2$ is executing instructions of $P_1$. Although $P_1$ will execute at $S$, its eligibility for that execution depends upon $P_1$'s attachment to $T_1$, $T_1$'s to $\pi_1$, and $\pi_1$'s to $P_1$. In order to deal with this difficulty, our

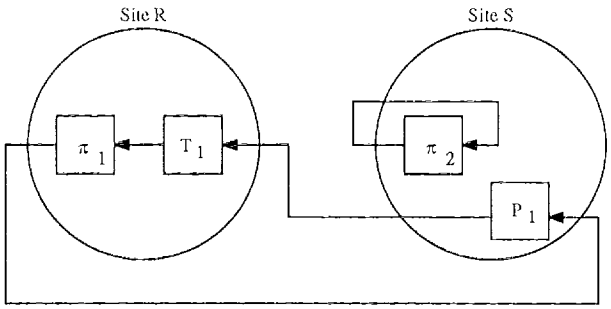



Figure 3: Scheduling Problem

implementation perverts the basic control model in two respects. Figure 4 illustrates that solution.

The solution makes use of *pseudo-tasks* and *remote links* in order to achieve the desired scheduling and execution. In Figure 4, $\tau_1$ is a pseudo-task which serves two purposes: it it a local representative of remote items above $P_1$ on it's operating chain, and it is a locally schedulable entity *on processor* $\pi_2$. A pseudo-task is constructed whenever an operating chain creates a subchain of itself on a remote processor. In the example, $T_1$ presumably performed a create/exchange pair in order to call $P_1$ at site $S$. If $P_1$ were to be terminated, $\tau_1$ would also be destroyed.

The implementation keeps *dlink*s strictly local. In order to define chains which extend across several processors, one employs remote links, *rlink*s. In defining the chains induced by the control model, an *rlink* takes precedence over a *dlink*. Thus, $\pi_1$, $T_1$, and $P_1$ still constitute the operating chain. However, if an element of $\Sigma$ has an *rlink*, its *dlink* must point to a pseudo-task. The *rlink* of an element of $\Pi$ merely serves to define chains in the model; the *dlink* of an

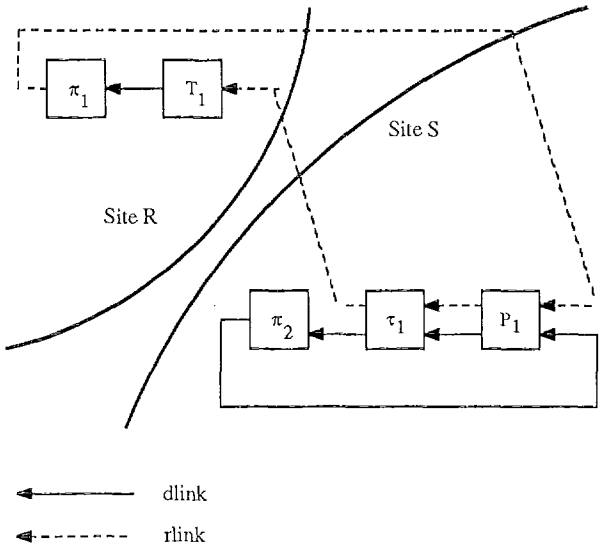



Figure 4: Scheduling Solution

element of Π identifies the instance consuming cycles on that processor. Under this convention, $\pi_1$ must be idle while $\pi_2$ is executing $P_1$. These policies are, of course, reflected in the actual implementation of the primitive control operations – **exchange** and **schedule** are considerably more complex in the implementation than in the model.

The interpreter daemons are interconnected with a TCP/IP virtual circuit between each pair of sites. Each daemon uses a specified well-known network address (port number). These decisions were made in order to get the prototype into execution quickly. We intend to implement a connectionless communication topology in the near future. An interpreter daemon attempts to receive a message under two circumstances – whenever there is no executable instance locally, and every fifth iteration through the main interpretation fetch-execute loop (the "fifth" policy was strictly *ad hoc*, dealing with messages on every iteration degraded performance markedly. Protocols involving fixed-length messages have been implemented for a number of purposes:

a. file transfer – Before execution begins, every site involved in the computation is sent a complete copy of the interpretable code trees. Switches which control the operation of the interpreter, such as the maximum length of a time slice, follow the code transfer.

b. remote instance creation – A message must be routed to the site at which an instance, be it task or non-task, is to be created. Allocation must be made at that site, and an instance value (actually the address of the activation record for that instance) must be returned. This implies a two-part representation of instance values, $< site, offset >$. The *site* component is node at which the instance is allocated; *offset* is the displacement from the beginning of the activation record storage structure on *site* at which the instance's activation record resides. In dealing with these pointers, special care is taken to short-circuit a good deal of interpreter code if the *site* is the local node. Note that this representation of instance values must be altered if and when instance migration is implemented.

c. control state alteration – Given that instances may be remote and that pointers may cross processor boundaries, control primitives obviously may require message traffic. Certain representation policies (the pseudo-task and a processor's "tail awareness" through its *rlink* mentioned above are prime examples) are implemented in order to alleviate message traffic.

d. non-local data reference – This is the obvious generalization of the policy of traversing slinks to resolve non-locals as described above.

e. mutual exclusion – In order to ensure atomicity of control operations, a technique similar to the Ricart and Agrawala distributed mutual exclusion algorithm[RA81] was implemented. The expansion of a control operation requests network mutex before the body of the operation; it releases mutex at the end of the operation. Suspension within a control operation releases mutex; activation will result in the reacquisition of mutex.

f. termination detection – The question of when the interpreter may actually terminate is relatively easy to deal with. We employ a straightforward token passing scheme on a Hamiltonian circuit of the sites similar to that described by Dijkstra *et al.*[DFv83]. This procedure is initiated by the master when it detects that it has no instances capable of execution. We make no distinction between a fully terminated program and one which is deadlocked (only idle chains exist).

We emphasize that the implementation is simply a prototype – the interpreter is currently evolving towards more general and efficient operation. The current system was implemented with the primary

goal of quickly obtaining a running system with reasonable performance so that the control abstraction facility could be investigated. Extensions to accommodate explicit message passing (as opposed to the system initiated message passing required to support the control model) and time-outs seem necessary to handle control regimes which are fault tolerant in any substantial way. Process migration is also a desired extension (the extraction of the "state" of an instance from the interpreter seems to be sufficiently cheap to permit easy migration (compare this with state extraction for a process on any reasonably complex operating system).

## 5. Concluding Remarks

We have found that CCAL provides a useful tool for investigating the characteristics and implementation of novel control forms. It also provides a rapid prototyping facility under which control forms most appropriate in a particular application domain may be developed. The facts that concurrency is a natural part of the language and that the implementation is distributed makes CCAL relevant for studying means of effectively constructing novel systems with desirable performance and reliability properties.

## Acknowledgements

We would like to acknowledge the input of our friends and colleagues, Mary Lou Soffa, Sarah Crall, and C. H. Lin, who have contributed to the work presented here.

## References

[Coo84] E. C. Cooper. Replicated procedure call. In *Proc. Third Annual ACM Symp. on Principles of Distributed Computing*, pages 220–232, 1984.

[DFv83] E. W. Dijkstra, W. H. J. Feijen, and A. J. M. vanGasteren. Derivation of a termination detection algorithm for distributed computations. *Information Processing Letters*, 16(5):217–219, 1983.

[Fis70] D. Fisher. *Control structures for programming languages.* PhD thesis, Department of Computer Science, Carnegie-Mellon University, 1970.

[HG78] D. R. Hanson and R. E. Griswold. The SL5 procedure mechanism. *Communications of the ACM*, 21(5):392–400, 1978.

[Hil83] P. Hilfinger. *Abstraction mechanisms and language design.* The MIT Press, 1983.

[KQ84] J. P. Kearns and D. Quammen. An efficient evaluation stack for ada tasking programs. In *IEEE Computer Society 1984 Conference on Ada Applications and Environments*, pages 33–40, 1984.

[KS83] J. P. Kearns and M. L. Soffa. The implementation of retention in a coroutine environment. *Acta Informatica*, 19:221–233, 1983.

[Lew79] B. Lewis. *Sequential control structure abstractions for programming languages.* Technical Report 79-10-02, Department of Computer Science, University of Washington, 1979.

[Qua86] D. Quammen. *Stack-based implementations of concurrent high level languages.* PhD thesis, Department of Computer Science, University of Pittsburgh, 1986.

[RA81] G. Ricart and A. K. Agrawala. An optimal algorithm for mutual exclusion in computer networks. *Communications of the ACM*, 24(1):9–17, 1981.

[THLZ77] L. Travis, M Honda, R. LeBlanc, and S. Zeigler. Design rationale for TELOS, a Pascal-based AI language. In *Proc. of the Symp. on Artificial Intelligence and Programming Languages*, pages 67–76, 1977. (Also published as ACM SIGPLAN Notices, Vol. 12(8).).

[Tur83] F. Turini. Abstractions of control environments. *BIT*, 23:21–35, 1983.

[Tur84] F. Turini. Magma2: a language oriented towards experiments in control. *ACM Transactions on Programming Languages and Systems*, 6(4):468–486, 1984.

[WD71] A. Wang and Ole-J. Dahl. Coroutine sequencing in a block-structured environment. *BIT*, 11:425–449, 1971.