THE DESCRIPTION AND CONTROL OF CHANGING PICTURES

Gregory F. Pfister
Department of Electrical Engineering and Computer Sciences
University of California, Berkeley, California

Abstract

DALI (Display Algorithm Language Interpreter) is a special-purpose programming language extension for the creation and control of changing pictures. DALI pictures are composed of elements called picture modules. These are analogous to procedural activations or processes, and contain arbitrary event-driven procedures called daemons.

Introduction

DALI (Display Algorithm Language Interpreter) is a high-level language extension for the creation and control of changing pictures, particularly pictures exhibiting complex static and dynamic interactions among their elements. Prior work on this subject has been performed in the development of computer graphics programming systems, e.g. |jl, 2, 3] . DALI differs strongly from such systems in that a DALI picture is not a passive data structure. Instead, a DALI picture is a structure of active elements akin to "processes", containing user-written procedures called daemons which are executed in changing the picture. The selection of daemons to run and the choice[1] of their order of execution is defined by a set of scheduling rules, built into the language, which are based on low-level functional relationships among daemons. A primary advantage of the DALI approach is modularity: each picture element locally describes its own behavior, i.e., manner of change.

Objects Defined by DALI

^ne base language extended by DALI wil1 be assumed here to be LISP. DALI adds to the base language four primary types of objects: outputs, daemons, picture modules (or just modules), and picture functions.

Outputs are locations where data can be stored, and are primarily used for communication. Outputs differ from "normal" storage locations in that a change in value of an output can be detected as an event by one or more active objects (i.e., daemons).

Daemons are parameterless procedures executed in response to some event. A typical event is a change in the value of one or more outputs. Since a daemon can alter the value of outputs in response to a change in other outputs, computed changes can be propagated through the entire picture. A daemon which can change a given output's value is said to specify that output; and a daemon which runs when a given output changes value is said to watch that output.

Picture modules are organizational units which contain daemons, outputs, and other picture modules in a hierarchical fashion. The objects contained in a picture module are said to be owned by that module. Picture modules also contain a local environment, a mapping from identifiers to values used only by the daemons owned by a module.

Picture functions are valued procedures written by the user. Application of a picture function creates, in lieu of a normal procedural activation, a picture module which remains in existence until explicitly deleted. The created picture module is the value of an application of a picture function. A picture module created by a daemon is owned by the daemon's owner; this creates a hierarchical structure analogous to the tree of procedure activations created in languages such as CONNIVER [4] and OREGANO [5J .

Picture functions are defined by an application of the function DEFPIC, as in

(DKFPIC name (argument list) body) ,

where name is the identifier naming the picture function. The argument list defines the local environment of the module created by the picture function. The body is a list of function applications which are evaluated as part of applying the picture function.

The structure created by applying picture functions, creating daemons, etc., is called the picture structure. Picture structures can be diagrammed as shown in Figures 1 and 2. The conventions used are: picture modules are cross-hatched circles, outputs are small open circles, and daemons are larger open circles; modules are connected to the modules they own by solid lines and to the daemons they own by dotted

outputs are connected to daemons which change their values by short solid lines; and dashed arrows lead from each output to all daemons which run when that output changes value.

Examples

Several examples of simple picture functions will now be presented.

The picture function RELP, which appears below, captures the notion of "relative position" in the following sense: it causes an output's value to be maintained as the sum of two other outputs' values.

```
(DEFPIC RELP (P1 P2 "OUT" SUM)
  (CONTIN (OUCH SUM (+ ,P1 ,P2)) ))
```

The picture structure created by applying RELP is shown in Figure 1.

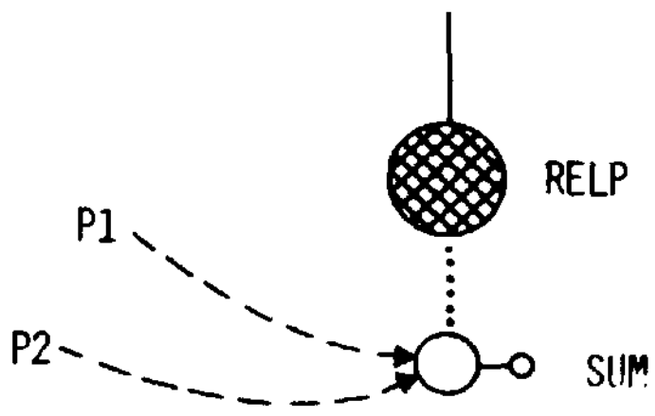


FIGURE 1

The above DEFPIC "declares" RELP to be a picture function which takes two arguments: P1 and P2; these must be outputs, due to the way in which RELP uses them. Applying RELP creates a module whose local environment holds P1, P2, and SUM; this is indicated by their presence in the argument list. The designator "OUT" indicates that at the time of RELP's application, SUM is to be initially bound to a newly created output with a value of NIL.

RELP's application of CONTIN creates a daemon which watches PI and P2, running when either or both change value. When this daemon runs, it will apply OUCH (Output value CHange) to change the value of the SUM output to the sum of the values of the outputs assigned to PI and P2. (Vector arithmetic is assumed.) The commas (,) appearing in the CONTIN daemon are syntactic sugar for an application of the function OVAL, which, applied to an output, returns the value of that output. In addition to creating the daemon, CONTIN also runs the body of the daemon once before returning, thus initializing SUM.

The designator "OUT", in addition to indicating an initialization of SUM, also indicates that the value of SUM can be obtained from the created picture module after RELP has returned. This access of RELP's local environment is performed by the function OUT, abbreviated by a prefixed exclamation point (!). Thus, for example,

```
(RELP DISP1 !(RELP DISP2 PT))
```

creates a "chain" of relative positions two elements long.

Now a picture function will be defined which manages a hardware display file entry causing a visible line to be drawn. This picture function, defined below as LINE, takes two arguments, both of which are outputs containing positions, and draws a line connecting its arguments' values.

```
(DEFPIC LINE (PI P2 "AUX" LINEID)
  (SFTQ LINEID (MAKE-ENTRY ,P1 ,P2))
  (AS-NEEDED
     (CHANGE-ENTRY LINEID ,PI ,P2) )
  (ONC DELETF
     (DESTROY-ENTRY LINEID))  )
```

The picture structure created by an application of LINE is shown in Figure 2.

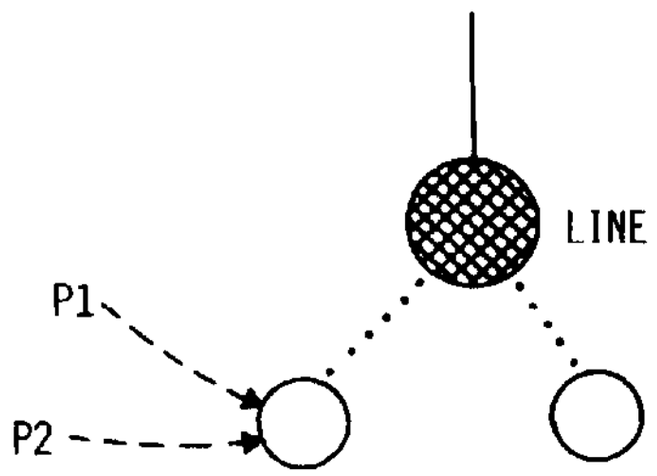


FIGURE 2

"AUX", in LINF's argument list, indicates that LINEID is an Auxiliary identifier which is to be retained in the local environment for use by daemons. LINEID holds a display-file-entry identifier returned by MAKE-ENTRY, which creates the display file entry. The application of AS-NEEDED creates a daemon like CONTIN but does not initially run that daemon. The created daemon watches PI and P2 and will apply the function

CHANGE-ENTRY to alter the display file entry whenever necessary. The application of ONC (for ON Condition) creates a daemon which is run as part of the process of deleting a LINE module; this daemon applies the function DESTRCY-ENTRY to deallocate the display file entry.

A relative line can now be defined as follows;

```
(DEFPIC RELINE (BASE DELTA)
  (LINE BASE !(RELP BASE DELTA)) )
```

RELINE creates no daemons; it effectively delegates the authority to handle the creation of the relative line to the modules it created. Such delegation need not be total. As an example, consider the picture function WATCHP3 below:

```
(DEFPIC WATCHP3 (P1 P2 P3
                 "AUXO" FILTER)
  (CONTIN
   (COND ((SAFE? ,P3)
          (OUCH FILTER ,P3))
         (T (DRASTIC-ACTION)) ))
  (LINE PI P2)
  (LINE P2 FILTER)
  (LINE FILTER P1) )
```

The "AUXO" argument list designator makes FILTER an AUXiliary identifier, like "AUX", but initializes FILTER to contain an output of value NIL.

WATCHP3 draws a triangle connecting its three argument outputs PI, P2, and P3, provided that P3 has some range of "safe" values as checked by the predicate SAFE?. If P3's value is not "safe", the daemon performs some "drastic action". It should be noted that the CONTIN daemon of WATCHP3 runs only when the value of P3 changes; changes to the values of P1 and P2 do not cause this daemon to run, but instead are responded to directly by daemons in the LINE modules created by WATCHP3.

The previous examples illustrate no mechanism by which outputs can be passed back through multiple levels of picture function invocation. For example, the output created by HELP in RELINE -- the computed endpoint of the line -- may be of interest to RELINE[1]s caller. This output passing is achieved via another argument list designator; "OUTU", for OUTput Uninitialized. "OUTU" operates like "OUT", in that the values of the identifier(s) it designates are receivable by using the function OUT (!); but "OUTU" does not initialize its designated identifiers, instead allowing then to be assigned values by the picture function body. A RELINE passing its computed endpoint back using "OUTU" can be coded as follows:

```
(DEFPIC RELINE1 (BASE DELTA "OUTU" EP)
  (SETQ EP !(RELP BASE DELTA))
  (LINE BASF EP))
```

Thus the fragment

```
(RELINE D1 !(RELINF1 D2 PT))
```

creates a 2-element chain of relative lines in which the embedded HELP daemons are directly connected. This is shown in Figure 3, which shows the picture structure created by the above fragment.

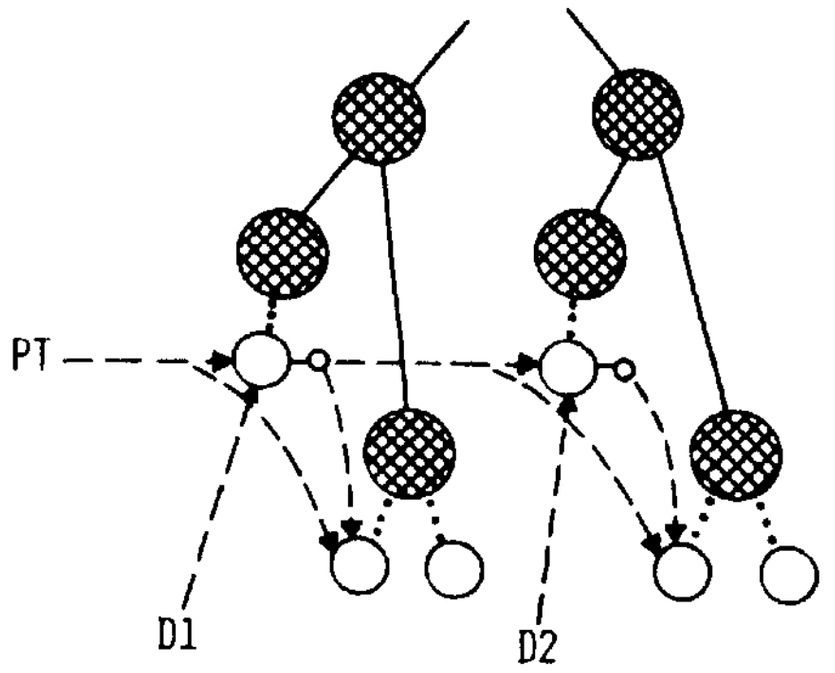


FIGURE 3

The Acyclic Daemon Scheduling Rules

The DALI daemon scheduling rules define which daemons are to be run and the order in which they are to be run. Here we will deal only with scheduling rules for a special case: the situation when inter-daemon relationships are acyclic, i.e., no daemon can change an output value and in so doing cause itself to be run again at some future time. Unrestricted – in particular, cyclic – relationships can be handled in DALI in a manner that leads to iteration, or "relaxation"; this more general situation is discussed in *[6]*, The acyclic rules described here form the basis of the treatment of the cyclic case, and occur in very many situations of interest.

Some definitions are needed prior to the presentation of the acyclic daemon scheduling rules:

A daemon A is the father of a daemon B if and only if A specifies an output watched by B.

A daemon A is an ancestor of a daemon B if and only if there exists a sequence of daemons D(0), D(l),. . . ,D(n) such that D(0)=A, D(n)=B, and for every i in the range 0<i<n+l, D(i-l) is a son of D(i).

A picture is said to be acyclic if and only if it contains no daemon which is its own ancestor.

The acyclic daemon scheduling rules

are:

Rule 1 (selection) A daemon will be run if and only if one or more of the outputs it watches is OUCHed; once run, it does not run again until such an OUCH occurs after its execution has terminated,

Rule 2 (noninterruption) Once a daemon D begins execution, no daemon ancestrally related to D may run until D terminates of its own accord.

Rule 3 (ancestors first) If daemons A and B are to be run, and A is a ancestor of B, then A is run before B.

Rule 4 (closure) If two daemons are to be run and they are not related in ancestry, they may run in any order.

There is a simple method for implementing the acyclic daemon scheduling rules on a single-processor system. Each daemon is assigned an unchanging integer priority at its creation according to the following rules: (X) the priority of the application program, considered as the "root daemon" ancestral to all other daemons, is 0; (2) the priority of a daemon is one greater than the largest of its fathers' priorities. Whenever an output is OUCHed, each daemon watching that output is immediately placed on a global daemon queue in order of increasing daemon priority. The running of daemons then consists of removing the daemon at the head of the queue from the queue and running it, repeating this until the queue is empty.

The scheduling rules above achieve two very important goals:

(1) Daemons can be considered as defining invariant relationships between their watched and specified outputs. This is critical to making DALI programs operate in a comprehensible manner. It means, for example, that wherever and whenever a RELP module is used, the SUM output can reliably be used in place of a computation of the sum of the module's inputs. This goal is primarily guaranteed by the ancestors first rule.

(2) Daemons operate in a stable environment. This is taken here to mean that the only data value changes which occur during that daemon's execution are performed by the daemon itself. This ensures that despite the fact that DALI is a multiple-environment, "multiple-process" system, each individual daemon can be written as if it were running on the simplest possible single program counter machine lacking any form of multiple processing or interrupts. Races, hazards, and other synchronization worries are defined out of existence from the start.

The "stable environment" and 'relationship" goals are as much a virtue as a necessity: a system like DALI, where use of hundreds of separate "processes" is implied, would literally be unusable if the "bugs" associated with coroutine or multiple-process systems could occur with their usual frequency. It should be noted that even given these goals, the ancestors first and closure rules do provide for the possibility of true parallel operation of daemons in a multiple-processor environment.

Environment Structure

The structure of the environment in which a DALI daemon operates is composed of four parts: the primitive environment, the global environment. the local environment and the temporary environment These are considered to be arranged in a stack, as shown in Figure 4.

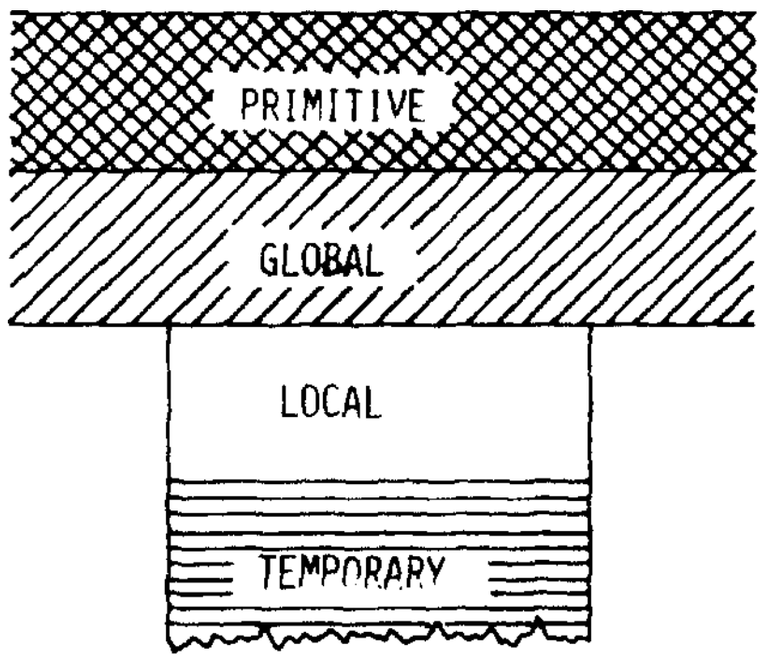


FIGURE 4

The primitive environment *consists* of those identifier-to-value mappings which are fixed for all time in the base language; it would typically include the identifier "+" in interpreted LISP, this environment is null.

The global environment contains identifier-to-value mappings which are defined by the user, and are generally static for the duration of an execution of a program. The names of procedures generally lie here. In LISP, this environment is unstructured; in a language with static block structure, such as ALGOL, the global environment would be structured in accordance with the static block structure.

A separate temporary environment is associated with each module, and retains its contents across daemon executions. The temporary environment contains purely temporary storage for use by daemons; it is empty when a daemon begins execution, and is emptied when the daemon finishes execution. Here we assume a stack-struc-

tured temporary environment; some modification of this organization would be necessary if FORTRAN, which is not stack-structured, were the base language.

The total environment structure existing when no daemon is executing, illustrated in Figure 5a (on a later page), is a broad but shallow tree with one temporary environment for each existing module. During any execution sequence that does not include picture function application, the total onvironment appears as shown in Figure 5b: one temporary environment exists, that of the currently running daemon. When picture functions are being applied, several temporary envirnoments can exist simultaneously in a nested fashion as shown in Figure 5c. This occurs because each picture function body executes as if it were a daemon.

From the viewpoint of physical storage allocation, the situation with the DALI environment structure is simpler than the above discussion makes it appear. While the local environments must be allocated out of a heap, they are created and destroyed realtively seldom and do not vary in size during execution. On the other hand, the much more rapidly varying temporary environments can be allocated in a single stack, since the only occurrence of multiple temporary environments is nested.

The sharp distinction DALI makes between local and temporary environments thus makes the environmental aspects of the language -- allocation of storage space and identifier lookup -- quite efficient.

Current Implementation

Like any other programming language, DALI cannot be considered truly "tested" until a body of users have applied it to their problems. Unfortunately, no usable implementation of DALI exists at this writing. An experimental version of DALI has been programmed using the language MUDDLE [7j on a Digital Equipment Corporation PDP-10 in the Programming Technology Group of M.I.T. Project MAC. This implementation was primarily intended to find out if the various pieces of DALI fit together in a reasonable manner, and to get an estimate of the size of the run-time system needed; the latter was gratifyingly small, amounting to approximately 6500 words of fairly "loose" PDP-10 program. However, this implementation is an interpreter written in an interpreter, and compares favorably in speed only with continental drift. Recent attempts at compiling the DALI interpreter have, however, produced sufficient gains in speed to justify the claims to efficiency made earlier in this paper.

Acknowledgements

The author would 1ike to thank Professor Michael Dertouzos of MIT lor his aid in the course of the work reported here, especially in untangling into a presentable dissertation the snarl of ideas that were DALI. Professor J.C. R. Licklider of MIT provided the computer resources needed for the experimental implementation of DALI. The work of Christopher Reeve and other members of the MAC Programming Technology group on MUDDLE and its compiler made the DALI interpreter possible. The research reported here was supported in part by Project MAC, an M.I.T. research program sponsored by the Advanced Research Projects Agency of the Department of Defense under Office of Naval Research Contract N00014-A-0362-0006.

References

1 Thomas, E. L., TENEX EfoS Display Software, Bolt, Beranek, and Newman, Inc., 50 Moulton St., Cambridge, MA

2 Rully, A. D., "A Subroutine Package for FORTRAN," IBM Systems Journal Vol. 7, Nos. 3 and 4, (1968) p. 248.

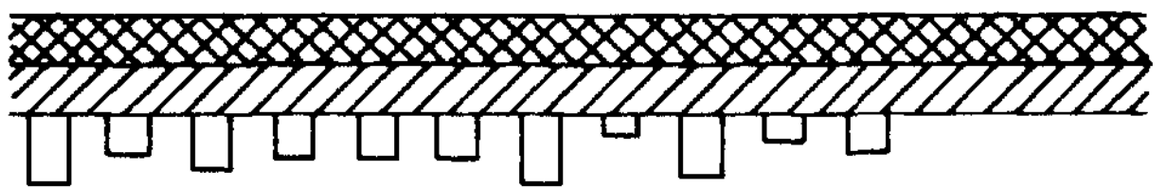
3 Christensen, C, Picson, E.E., "Multi-Function Graphics for a Large Computer System", 1967 Fall Joint Computer Conference, Thompson Books, Washington, D.C., pp. 697 ff.

4 McDermott, D., and Sussman, G. J., The CONNIVER Reference Manual, MIT Artificial Intelligence Laboratory Memo 259, May 1972

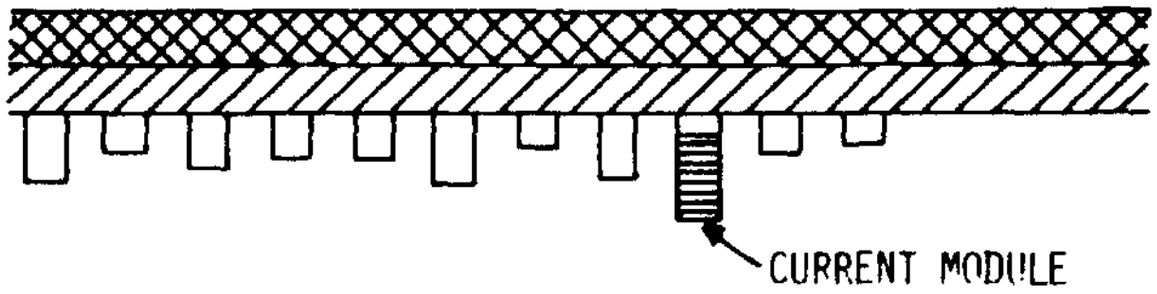
5 Berry, D. M., "Introduction to OREGANO", Proceedings of a_ Symposium on Data Structures in Programming Languages. ACM SIGPLAN Notices Vol. 6 No. 2 (Feb. 1971) pp. 171-190.

6 Pfister, G. F., The Computer Control of Changing Pictures, MIT Project MAC Technical Report TR-135, Project MAC, MIT, Cambridge, MA, 1974.

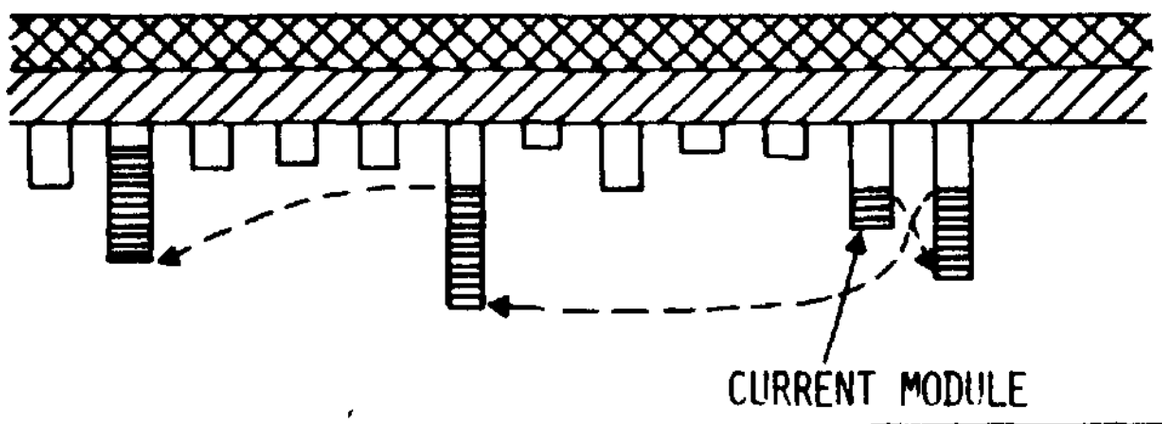
7 Pfister, G.F., A MUDDLE Primer, Project MAC Programming Technology Group document SYS.11.01, Dec. 1972, Project MAC, Cambridge, MA



A QUIESCENT STATE



B NON-ADDITIVE CHANGE



C ADDITIVE CHANGE

FIGURE 5