CMU-CS-82-129

DPL-82: A LANGUAGE FOR DISTRIBUTED PROCESSING

Lars Warren Ericson
Computer Science Department
Carnegie-Mellon University
Pittsburgh, PA 15213

26 July 1982

To be published in the Proceedings of the IEEE Third International Conference on Distributed Systems, October, 1982, Ft. Lauderdale, Florida

Abstract

DPL-82 is a language for composing programs of concurrently-executing processes. Processes may be all on a single machine or may be distributed over a set of processors connected to a network. The semantics of the language is derived from the underlying interprocess communication facility (IPC) and from the dataflow model of computation. This paper discusses the major concepts of the language, namely *nodes*, *arcs*, *connections*, *tokens*, *signals*, and *activations*, and presents examples which illustrate the construction of distributed programs in DPL-82 with *internal* arcs, *external* arcs and *child* arcs. Features for process-to-processor mapping and dead process restart are mentioned. The paper concludes with some ideas for future research.

This research was sponsored by the Defense Advanced Research Projects Agency (DOD), ARPA Order No. 3597, monitored by the Air Force Avionics Laboratory under Contract F33615-78-C-1551. The views and conclusions contained in this document are those of the author and should not be interpreted as representing official policies, either expressed or implied, of the Defense Advanced Research Projects Agency or the US Government.

REFERENCES

Table of Contents

List of Figures

1. INTRODUCTION

The intention of the research leading to DPL-82 was to implement a programming language to control a local network of computers as if they were a single computing engine. There are many schools of thought with respect to the choice of an underlying mechanism for control and communication in a distributed program. One important decision is whether the flow of control or the flow of data is emphasized. The control-flow end of the spectrum is characterized in the *remote procedure call* concept.[1] Somewhere in the middle of the control/data-emphasis spectrum is the MIT Actor model of computation,[2] whose underlying metaphor is the notation of *continuation.* Current Actor language implementation research is embodied in Lieberman's language ACT1,[3, 4] and research is also being done by Clinger on the denotational semantics of Actors, which includes an Actor language called ATOLIA.[5] The data-flow end of the spectrum is characterized in the *dataflow* concept.[6, 7, 8] An important descriptive method arising from dataflow research is the UCLA SARA Graph Model of behavior.[9] Our language, DPL-82, is a dataflow language. A final, important, distinction is whether control "remains centralized" after the initial stages of execution of the distributed program. This tends to be the case in the majority of approaches cited in this paper; the exception that proves the rule is the *worm control program* research of Shoch.[10]*

DPL-82 derives its emphasis, not from theoretical constructions of language features for parallelism or out of designs for parallel machines, but from research on operating system mechanisms for interprocess communication. DPL-82 depends on the the *port* and *message* concepts of Rashid's CMU VAX/UNIX interprocess communication facility (referred to in this paper as the IPC),[11] to implement a interprocess communication path concept called the *arc.* The *arc* is very similar to Morrison's *data stream linkage mechanism,*[12, 13] and the *window* concept of the Honeywell HXDP operating system.[14, 15] A language very similar to DPL-82 is Lesser's PCL.[16]

There are several veins of distributed processing language research which do not fall directly into the classifications given above. One is that of languages written for specific hardware architectures, such as Dannenberg's AMPL for the CMU CM* processor[17] and Snodgrass's object-oriented language COLA for the CMU C.MMP processor.[18] There is much work centered around the ideas of *tasking* and *semaphores*, such as the ADA tasking facility,[19] STAROS Task Force[20] and CLU *guardians.*[21] Finally we must mention Hoare's very popular CSP language.[22] CSP has a process concept and a compact notation for linking the inputs and outputs of processes, but unlike most of the work arising out of the experimental systems-building community, emphasizes processes which are very small computationally, and binds processes very tightly to each other (each process description is written for exactly one named caller, hence there is no possibility for "libraries" of

*Unfortunately that research, which was an experimental exercise in distributed system building, has not, to our knowledge, been abstracted into a set of high-level "worm control structures" that could be compactly integrated into a language.

processes which could be "linked" together to form a distributed program).

In DPL-82 a distributed program is composed of processes executing on a number (possibly one) of machines. Each machine in the network supplies some number of processes. A process is a running core image, and each computer might store on secondary memory executable core images for some subset of the different kinds of processes which compose the entire program. The processes are connected with communication channels supplied by the IPC.* To make a DPL-82 distributed program or sub-program, user supplied PASCAL code is embedded in a process description which defines the communications interface (input and output communication paths) for the process being described, subprocess requirements, and interconnection of communication paths of subprocess. This description is translated into a complete PASCAL program for that process which may then be compiled and executed in the DPL-82 runtime environment. The runtime environment supplies protocols via the IPC for establishing communication paths between subprocesses and (network transparent) passing of data on those paths, and a facility to allow a DPL-82 distributed program component (process) on one machine to request the loading of a process on another machine (which process may in turn cause the loading of additional processes, etc). DPL-82 also provides the ability to pass parameters to subprocesses at subprocess load-time. For example, the size of various portions of a distributed program (in terms of number of processes) may be a runtime-computed function of such parameters.

2. PARTICULARS OF THE LANGUAGE

The processes that make up a DPL-82 distributed program are called *nodes*. Nodes are *not* to be construed as processors in a network, but rather as processes that those processors provide. A node description consists of a number of sections:

- The name of the node.
- Declaration of its communication links (*arcs*): the `internal_arcs`, `external_arcs` and `child_arcs` sections**.
- Declaration of (*children nodes*): the `uses` section.
- Child startup and initialization parameters: the `initialize` section.
- Arc interconnection: the `connect` section which contains arc-to-arc pairs marked by arrows (=>).

---

*A process cannot tell whether a given connection is to another process in the same machine or to a process on another machine on the network. This is a property of the IPC mechanism. DPL-82 is, in effect, a programming language made out of a set of protocols and capabilities dependent on this particular interprocess communication mechanism.

**code and code are set in this typeface.

- Startup-time-only computation: the `to_instantiate` section, and
- *activation conditions*: the `activate` section which controls the handling of tokens, signals and child processes thereafter.

2.1. Arcs

The communication links between nodes are called *arcs*. Arcs are implemented with IPC ports.* A node, through its DPL-82 description, may declare a variety of *input* and *output* arcs. Arcs transmit *tokens* and *signals*. A *token*, modelled after the concept of a token in dataflow networks,[6] is a typed data object. A *signal* is a string. Tokens and signals are communicated by IPC messages.

There are three sorts of arcs:

- An *internal arc* is a connection between a parent node and a child node. The actual connection is made by the parent. The *parent* is the process which starts a given (*child*) process, whether or not the parent process is on the same machine as the child process.
- An *external arc* is a path into or out of a node, which is declared inside that node, but whose connections are set by the parent of the node.
- A *child arc* is declared by a child node, and is connected to a sibling of the child by the parent node, but is then subsequently reconnected by the child to one of *its* children. Tokens or signals subsequently passing along this connection do not go to the child, but go directly to the child's child from the originator of the arc or signal.

These three varieties of input and output arcs provide distributed processing analogues to expressions, subroutines and main programs in uniprocessing languages. A computational process with external arcs is like an expression.** A child process which configures a sub-network of interconnected nodes and associates unconnected arcs of the sub-network with child arcs is like a subroutine. Finally, a node with no external arcs that creates a network of child nodes (and may connect itself to that network with internal arcs) is like a main program.

2.2. Starting nodes

The `initialize` statement loads a process and provides it with parameters. For example:

```
(for i from 1 to 5 (initialize node TrafficLight(i)[i]))
```

starts up five `TrafficLights`. The first use of `i` (in parentheses) refers to the particular instance of

---

*A *port* is like a mailbox, of which a process may own several, to which data may be sent in the form of *messages*.

**This analogy is fairly apt, given work such as that of Arvind which takes expressions in a functional language and "flattens" them into dataflow networks.[23]

`TrafficLight` being loaded. The second use (in brackets) is a integer-valued parameter being passed to that `TrafficLight` instance.*

One can start a process on another machine with a statement like:

```
(initialize node Correlator on_host "CAD-VAX")
```

which says to load a process whose core image is stored on file name `Correlator` on the processor whose symbolic local net address is `"CAD-VAX"`.**

2.3. Connecting nodes

The => statement makes connections between nodes. For example:

```
(=> philosopher[4]:hand fork[5]:handle)
```

connects `philosopher` number four's `hand` to `fork` number five's `handle`.

2.4. Activation conditions

The remainder of the node description is taken up by the *activation conditions*, which allow tokens and signals to be received and transmitted, and other actions to be taken such as node self-termination and child node restarting. These may be contingent upon signal and token arrival, timeouts, internal state of a user's PASCAL code, and boolean combinations of the above.

The following DPL-82 activation condition-action pair detects the death of a process and restarts it:

```
((is_dead some_node)(restart some_node))
```

This is a primitive restarting capability, whose only effect is to start a process of the same name and give it the connections of the dead process***. Nothing in the language deals with the question of restoring the internal state of a lost process.

3. SOME EXAMPLES

We will now present a number of simple nodes to illustrate the concepts discussed above. It is important to note that these nodes are not very computation-intensive, and that the real economy of this language comes with nodes which do more work. Also, we will not, in this paper, present any timing measurements for

---

*We are passing it because we want each `TrafficLight` to know what it's name is with respect to the parent. A child node does not automatically know it's "name" in DPL-82.

**Dannenberg discusses a design for a more general *network operating system* mechanism called the BUTLER for remotely allocating processes.[24]

***The ability of connections to survive process death is a property of the IPC mechanism.

establishing connections or passing tokens or signals, and we will not consider the problems of optimal process-to-processor mapping. This is partly due to space limitations, and partly due to the fact that the primary thrust of this research has been to achieve correct mechanisms in terms of functionality and linguistic expression. Future research, on a successor to DPL-82 (DPL-83?), may be concerned with mechanism optimization and resource allocation issues.*

3.5. Nodes with inputs and outputs

The simplest kind of interesting node must communicate with and start other concurrently processing nodes in order to perform a computation. To do this, the node must declare input and output arcs and must include in its description specific commands for starting up other nodes. Let us concieve of a node which performs the x^2 function called `xsquared`, and another node which utilizes `xsquared` in a simple parallel computation, which we will call `plus2xsq`. It will present two `xsquared` nodes with numbers, and sum and print their results.

```
(node xsquared
        external_arcs ((integer inx) => (integer outx))
        procedure xsq (inx: integer; var outx: integer);
        begin
                Writeln("Toto, inx is ", inx:1, "!");
                outx := inx * inx;
        end;
        activate ((tokens_available)
                  (apply xsq to [inx, outx])
                  (terminate)))
```

*Nelson and Spector have worked on low-level optimizations of remote procedure call, an alternative method of structuring distributed computations.[1, 25] Bryant, Chu and Arvind have researched the issues of distributed computer resource allocation from the operating system point of view, and the related issue of the optimal process-to-processor mapping from the program's point of view.[26, 27, 23]

```
(node plus2xsq
        internal_arcs
                ((integer result1 result2) =>
                 (integer x1 x2))
        procedure plusx12 (var x1, x2: integer);
        begin x1 := 3; x2 := 4; end;
        procedure getresults (r1, r2: integer);
        begin
              writeln("The result is ", (r1+r2):1, ".");
        end;
        uses    (uses array[2 max 2] of node xsquared)
        initialize
                (initialize node xsquared[1])
                (initialize node xsquared[2])
        connect (=> x1 xsquared[1]:inx)
                (=> x2 xsquared[2]:inx)
                (=> xsquared[1]:outx result1)
                (=> xsquared[2]:outx result2)
        to_instantiate (apply plusx12 to [x1, x2])
        activate ((tokens_available)
                  (apply getresults
                        to [result1, result2])
                  (terminate)))
```

This may be pictured as in Figure 3-1.

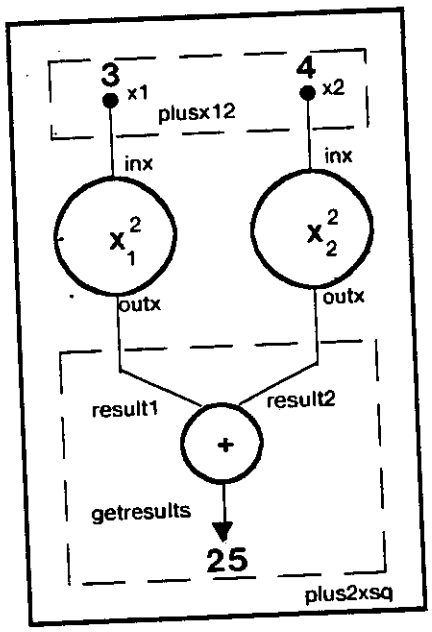

Figure 3-1: A node illustrating internal arcs

When executed, `plus2xsq` gives the following results:

```
$ plus2xsq
Toto, inx is 4!
Toto, inx is 3!
The result is 25.
```

The node that started the node that owns a given set of input arcs does not know about these internal arcs. The only way that information can flow out of the purview of the internal arc owner and into the purview of the node that starts up the owner, is if the internal arc owner connects a child arc to an internal arc (see below).

We can illustrate the use of child arcs with the node `xfourth`, which has one input child arc and one output child arc. `xfourth` outputs the value of the input raised to the fourth power. `xfourth` is shown in Figure 3-2.

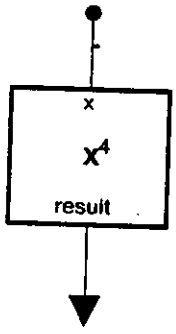

Figure 3-2: A node illustrating child arcs

The code for `xfourth`, like a subroutine, hides the implementation of the arithmetic operation as a subnetwork of concurrently executing nodes. This subnetwork is analogous to the lines of code that define a subroutine body.

```
(node xfourth
        child_arcs ((integer x) => (integer result))
        var        root: string; sub: integer;
        uses       (uses array[2 max 2]
                        of node newxsquared)

        initialize
                (initialize node newxsquared[1])
                (initialize node newxsquared[2])
        connect (=> x newxsquared[1]:inx)
                (=> newxsquared[1]:outx
                    newxsquared[2]:inx)
                (=> result newxsquared[2]:outx)
        activate ((signal_from parent halt)
                  (slow signal_to children
                        [newxsquared[1],newxsquared[2]]
                        halt)
                  (terminate)))
```

We can exercise `xfourth` with the following node, `fourthstream`, which reads integers from the terminal and prints out their fourth powers. When the end of the input stream is reached, `fourthstream` terminates after sending a `halt` signal to `xfourth`. `xfourth` will then signal its `xsquared` subordinates to `halt`, and terminate itself. The `xsquared` nodes will terminate themselves, and the distributed computation will conclude.

The definition of `xsquared` must be modified slightly to catch the signal. We will call the new version `newxsquared`. `newxsquared` also does not terminate after the first set of input tokens, but rather cycles indefinitely until the `halt` signal has been received.

3.6. Cyclic checking of activation conditions

The concept of *cycling* is very important. The default action of the node is to wait for IPC messages which represent tokens or signals, then evaluate the activation conditions, which usually refer to token or signal arrival events. However note that in `fourthstream` the first activation condition refers to a side-effect generated by the next activation conditions. This side-effect emanates from the `input_stream` procedure, and signals that the end of input has been reached. The creation of the side-effect is dependent on the user's typein, and not on message events (although it strictly follows the last `result` token arrival). Hence the detection of the side-effect does not involve waiting for a message event. If we chose the default action of waiting forever for a message event, then the first activation condition would *never* be tested.

The solution chosen here is to modify the amount of time we are willing to wait for message events. This time is chosen so that we don't needlessly check the activation conditions while actually waiting for messages,

and so that we don't wait too long, when we aren't expecting messages, to get around to checking the non-message-event-related activation conditions. When the user is typing in numbers, `fourthstream` might catch a "timeout" or two before receiving the response, but this is harmless. When the user types in the number -1, indicating end-of-input, the variable `EOS` is changed to `true`. The node will wait at most 500 milliseconds, when `EOS` condition is true, to signal "end of computation" and terminate.*

```
(node newxsquared
        external_arcs ((integer inx) => (integer outx))
        procedure xsq (inx: integer; var outx: integer);
        begin outx := inx * inx;   end;
        activate ((tokens_available)
                   (apply xsq to [inx, outx]))
                 ((signal_from parent halt)(terminate)))
```

```
(node fourthstream
        internal_arcs ((integer result) => (integer x))
        var EOS: boolean;
        function input_stream (var x1: integer):boolean;
        begin write(">"); readln(x1);
              if (x1 = -1)
              then begin EOS :=true;
                         input_stream := false;
                   end
              else input_stream := true;
        end;
        procedure init_stream (var x: integer);
        var toss: boolean;
        begin EOS := FALSE;
              toss := input_stream(x);
        end;
        procedure output_stream (result: integer);
        begin
              writeln("The result is ", result:1, ".");
        end;
        uses       (uses node xfourth)
        initialize (initialize node xfourth)
        connect    (=> x xfourth:x)
                   (=> xfourth:result result)
        to_instantiate (apply init_stream to [x])
        cycle_time_is (= 500)
        activate ((= EOS)
                  (slow signal_to children
                        [xfourth] halt)
                  (terminate))
                 ((tokens_available)
                  (apply output_stream to [result])
                  (test input_stream [] ? [x] : [])))
```

We have now used child, external and internal arcs. `fourthstream`'s execution network is pictured in Figure 3-3. The following is a sample of how it behaves:

```
$ fourthstream
> 2
The result is 16.
> 5
The result is 625.
> -1
$
```

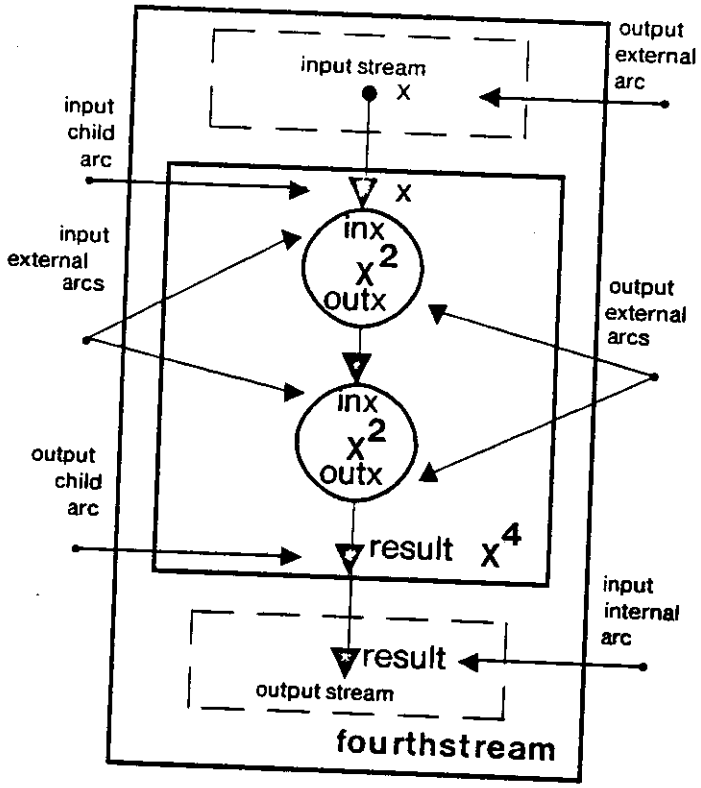

Figure 3-3: A network of nodes with external, internal and child arcs

4. FUTURE RESEARCH

We intend to implement abstractions of the features of this language in Edinburgh ML,[29] an interpreted, typed functional language designed for research in denotational semantics,[30] as a first step toward the development of a power-domain based[31] semantics of the underlying IPC facility and the conceptual features of the language.

There are some intriguing possibilities for the design of a microprocessor or microcoding of a processor whose instruction set is optimized towards message-passing and distributed programs whose underlying control construct is the *continuation*. The notion of a continuation is to be found in Actor semantics,[2] the notion of the RTRANSFER in remote procedure call,[1] and in denotational semantics.[32]

The concept of *constraint networks* may find a home in the distributed processing context with dataflow-like languages, such as a successor to DPL-82, which are extended to include the notion of bi-directional arcs (now simulatable with pairs of input and output arcs) and appropriate relaxation procedures.[33, 34]

5. ACKNOWLEDGEMENTS

The author is indebted to Prof. Raj Reddy for providing him with funding and research facilities to do this work. Professors Richard Rashid and Dana Scott have provided an enormous amount of support, encouragement and constructive criticism. Conversations with David Hornig, Roger Dannenberg, and other members of CMU's distributed systems and distributed sensor network communities have also been very helpful. Jeff Shrager has tirelessly reviewed many drafts of this paper: please give him a teddy bear if you find him.

REFERENCES

1. Nelson, Bruce Jay, *Remote Procedure Call*, PhD dissertation, CMU CSD, May 1981.

2. Hewitt, C., "Viewing Control Structures as Patterns of Passing Messages," *Artificial Intelligence*, Vol. 8, No. 3, January, 1977.

3. Lieberman, Henry, "A Preview of Act 1," AI Memo 625, MIT AI Laboratory, April 1981.

4. Lieberman, Henry, "Thinking About Lots Of Things At Once Without Getting Confused: Parallelism in Act 1," AI Memo 626, MIT AI Laboratory, May 1981.

5. Clinger, William Douglas, *Foundations of Actor Semantics*, PhD dissertation, MIT, May 1981.

6. Dennis, Jack B., "Data Flow Supercomputers," *IEEE Computer*, November, 1980.

7. Dennis, Jack B., et. al., "Research Directions in Computer Architecture," MIT LCS Tech Report TM-114, MIT, September 1978.

8. Gurd, John and Ian Watson, "Data Driven System for High Speed Parallel Computing," *Computer Design*, June-July, 1980.

9. Ruggiero, W., et. al., "Analysis of data flow models using the SARA graph model of behavior," *Proceedings of the AFIPS National Computer Conference*, 1979.

10. Shoch, J. F. and J.A. Hupp, "Notes on the *Worm* programs -- some early experience with a distributed computation," Tech. report SSL-80-3, Xerox PARC, September 1980.

11. Rashid, Richard F., "An Inter-Process Communication Facility for UNIX," Tech. report CMU-CS-80-124, CMU CSD, February 1980.

12. Morrison, J.P., "Data Stream Linkage Mechanism," *IBM Systems Journal*, Vol. 17, No. 4, 1978.

13. Levine, J.R. and J.P. Morrison, "Forum: Data stream linkage and the UNIX system," *IBM Systems Journal*, Vol. 18, No. 3, 1979.

14. Boebert, W.E., "The HXDP Executive Interim Report," Tech. report 78SRC53, Honeywell Systems & Research Center, June 1978.

15. Boebert, W. E., "Concepts and Facilities of the HXDP Executive," Tech. report 78SRC21, Honeywell Systems & Research Ctr., March 1978.

16. Lesser, Victor, Daniel Serrain and Jeff Bonar, "PCL: A Process-Oriented Job Control Language," *Proceedings of the 1st Intl. Conf. on Distributed Computing Systems*, IEEE, 1979.

17. Dannenberg, Roger B., "AMPL: Design, Implementation and Evaluation of A Multiprocessing Language," Tech. report, CMU Computer Science Dept., March 1981.

18. Snodgrass, Richard, "An Object-Oriented Command Language," Tech. report CMU-CS-80-146, CMU CSD, October 1980.

19. Department of Defense, *Reference Manual for the ADA Programming Language*, 1980.

20. Jones, Anita K. and Karsten Schwans, "TASK Forces: Distributed Software for Solving Problems of Substantial Size," *Proceedings of 4th IEEE Software Engineering Conference*, September 1979, pp. 315-330.

21. Liskov, Barbara, "Linguistic Support for Distributed Programs: A Status Report," Memo 201, MIT LCS Computation Structures Group, October 1980.

22. Hoare, C.A.R., "Communicating Sequential Processes," *Communications of the ACM*, Vol. 21, No. 8, 1978.

23. Arvind, "Decomposing a Program for Multiple Procesor Systems," *Proceedings of the 1980 International Conference on Parallel Processing*, IEEE, 1980.

24. Dannenberg, Roger B., "The Spice Butler," Spice Project Internal Working Paper Spice Document S110, CMU CSD Spice Project, August 1981.

25. Spector, Alfred Z., "Performing Remote Operations Efficiently on a Local Computer Network," Tech. report STANCS-80-850, Stanford University Computer Science Dept., December 1980.

26. Bryant, R.M., and R.A. Finkel, "A Stable Distributed Scheduling Algorithm", University of Wisconsin Computer Science Dept.

27. Chu, W.W., L.J. Holloway, M. Lan and E. Kemal, "Task Allocation in Distributed Data Processing," *IEEE Computer*, November, 1980.

28. Gostelow, K., et. al., "Proper Termination of Flow-Of-Control in Programs Involving Concurrent Processes," *Proceedings of the 1972 ACM National Computer Conference*, ACM, 1972.

29. Gordon, M. J., A.J. Milner and C.P. Wadsworth, *Edinburgh LCF*, Springer-Verlag, Lecture Notes in Computer Science, Vol. 78, 1979.

30. Gordon, Michael J.C., *The Denotational Description of Programming Languages*, Springer-Verlag, New York, 1979.

31. Stoy, Joseph E., *Denotational Semantics: the Scott-Strachey Approach to Programming Language Theory*, MIT Press, Cambridge, Mass., 1977.

32. Sethi, Ravi and Adrian Tang, "Constructing Call-by-Value Continuation Semantics," *Journal of the ACM*, Vol. 27, No. 3, July, 1980, pp. 580-597.

33. Steele, Guy L., "The Definition and Implementation of a Computer Programming Language Based on

Constraints," Tech. report AI-TR-595, MIT AI Laboratory, August 1980.

34. Borning, A., "ThingLab: a constraint-oriented simulation laboratory," Tech. report STAN-CS-79-746, Stanford University CSD, July 1979.