A Methodology for Debugging Data Flow Programs

Computation Structures Group Memo 219
12 December 1981
(Revised 15 October 1982)

Nena B. Bauman[1]

Robert A. Iannucci[2]

[1]Research support provided by the National Science Foundation under grant MCS-7915255.

[2]Research support provided by the International Business Machines Corporation.

Abstract

Data flow languages, being relatively new on the scene, have very little in the way of generally--accepted programming methodologies. Most researchers still tend to think of operating data flow machines in a sort of *batch* mode; that is, programs are compiled for later execution. The execution takes place in a *static* environment; the running program does not change. Debugging has been viewed as an activity carried out externally. Moreover, symbolic debugging is inherently difficult due to the nature of data flow. Such languages are functional and, therefore, have no states in the conventional sense. This is touted as a strength of data flow languages but becomes a hindrance when program debugging is viewed in a traditional context. This paper presents an alternative position.

We focus on the fundamental issues of "debuggability" as they relate to data flow. While superficially it appears that the data flow paradigm is doing everything in its power to hinder interactive debugging, many characteristics of data flow can actually be used to advantage. We exploit the differences.

Key words and phrases: data flow, debugging, functional languages, programming methodology

Table of Contents

List of Figures

A Methodology for Debugging Data Flow Programs

1. Introduction

In the last ten years, a serious look has been taken at the fundamental model of computation that underlies most systems in use today [4]. This model (proposed by von Neumann in the 1940s) is characterized by a *central processing unit* communicating in a more or less serial fashion with a *memory subsystem.* This model describes computation as the sequential following of a list of instructions (stored in the memory subsystem); these instructions are requested by the central processing unit, and are returned via the serial communication path (we use the term *serial* to describe a path which can carry only one datum at a time).

This deceptively simple set of assumptions about how computers should be constructed has had a profound influence on the way we think about them. The von Neumann paradigm gives rise to two very undesirable characteristics of program behavior:

1. **Sequential Execution:** The von Neumann model is, by definition, a sequential processing of instructions.

2. **Shared, or "Global" Storage:** This manifests itself in many ways and has an implicit sequentializing effect on programming (*e.g.*, shared mutable data).

Data flow [3, 6] has been suggested as an alternative model of computation. In such an architecture, rather than fetching data upon the availability of instructions (*i.e.*, when the program counter points to the instruction), *instructions* are fetched upon the availability of data (sometimes referred to as data driven computation). Data flow has the potential to exploit the concurrency available in highly parallel algorithms. To do this, however, the languages used to describe the algorithms must be built around the concept of freedom from side-effects.

The concept of *debugging* (as a consequence of inherently flawed human systems) is not well understood in the context of von Neumann systems. Unfortunately, even the small amount of understanding we have of debugging on a von Neumann system is not directly applicable to data flow systems; von Neumann debuggers work with sequential flow of control and global memory, whereas data flow prohibits both of these. Hence, we must develop a better, and more fundamental, understanding of what debugging is[1].

[1]We recognize the need to provide a hierarchy of debugging tools. This paper deals with that level of the hierarchy that is associated with the construction of programs written in high level data flow languages.

2. A Bug is a Bug is a Bug

The following terms are used in the sequel:

- **Bug:** That which causes the behavior of a program or system to differ from its intended behavior.

- **Debug:** The methodical process of identifying *bugs* and their causes; frequently followed by appropriate modification of the associated program or system to remove these.

- **Erroneous:** Something which is incorrect or incomplete.

- **Specification:** The formal description of the intended behavior of a program or system. Such a description, to be considered complete, must describe said behavior under all possible conditions.

- **Debugging Device:** A powerful and flexible execution monitoring mechanism.

Bugs come in many varieties and originate in many different ways, most of which can be traced back to the actions of one or more humans. Bugs may arise from one or more of the following: erroneous specifications, erroneous implementation of the specifications, erroneous encoding of the algorithms into the programming language being used, failure of the language translation/checking process, failure of the hardware, or failure of the bug detection mechanism (nonexistent and undetected bugs).

The obvious approach to debugging would thus be to eliminate all sources of bugs. While this is a noble goal, it is inherently unachievable due to the assumption that we, the flawed humans, would be able to correctly identify and eliminate all the *other* flaws that we have introduced. With this disconcerting fact in mind, we turn from the problem of bug elimination to the problem of bug reduction.

Traditional approaches to bug reduction are part of what we call *good programming methodologies.* The techniques include

1. Creation of a specification which is as complete as possible.

2. Careful algorithm development/proof.

3. Careful encoding in a programming language:

    a. Selection of an appropriate language/proof system.

    b. Compile-time checking (where appropriate).

    c. Run-time checking (where appropriate).

Only when all else fails should the programmer resort to explicit debugging. Realistically, this is unavoidable in some situations. Where it is appropriate, debugging tools should be made available to assist in this process. It is very important to stress that any such tools should allow the programmer to deal in the concepts with which he is familiar - at best, we believe, he should be able to use the full power of the language in which the program is written during the debugging process (reading a core dump does not adhere to this principle).

More often than not, a programmer will rely on his intuition and experience to decide on a starting point when tracking down a bug. This usually begins with a probabilistic determination of the most likely cause. A good debugging environment should not stand in the programmer's way in making this choice. Ideally, the debugger could provide suggestions based on program state, exception conditions reflected at a lower level, and so on.

3. The Problems of Debugging Data Flow Programs

The last section discussed bugs and debugging in an architecture independent context. Unfortunately, our debugging intuition has been heavily biased towards von Neumann ideas. It is, therefore, important to re-describe the problem of debugging in a data flow context.

3.1. The Data Flow Computation Model

Programs written in data flow languages are somewhat like other modern programming languages. The two data flow languages in use at M.I.T., VAL (Value-Oriented Algorithmic Language) and ID (Irvine Dataflow), have guided our thinking. Examples are based on these languages. The concepts presented here, however, may be applied to languages developed with the same goals in mind (*i.e.*, executing on a data driven machine).[2]

A discussion of data flow languages is given in [1]. The salient features of any data flow language are:

1. **Freedom from side-effects:** This is a characteristic of applicative (or functional) languages such as pure LISP [8] or Backus' FP [4].

2. **Data driven computation:** Instructions are ready for execution when their operands are available. This establishes strict equivalence between instruction sequencing and the data dependencies inherent in the program.

3. **Single assignment:** These languages are *definitional* rather than *imperative.*

Data flow programs are compiled into directed graph form, making the explicit data

[2]VAL is a strongly typed language, developed for use in application areas with numerical computation that strain the limits of high performance machines [2]. ID is a non-strongly typed language, developed for general purpose use.

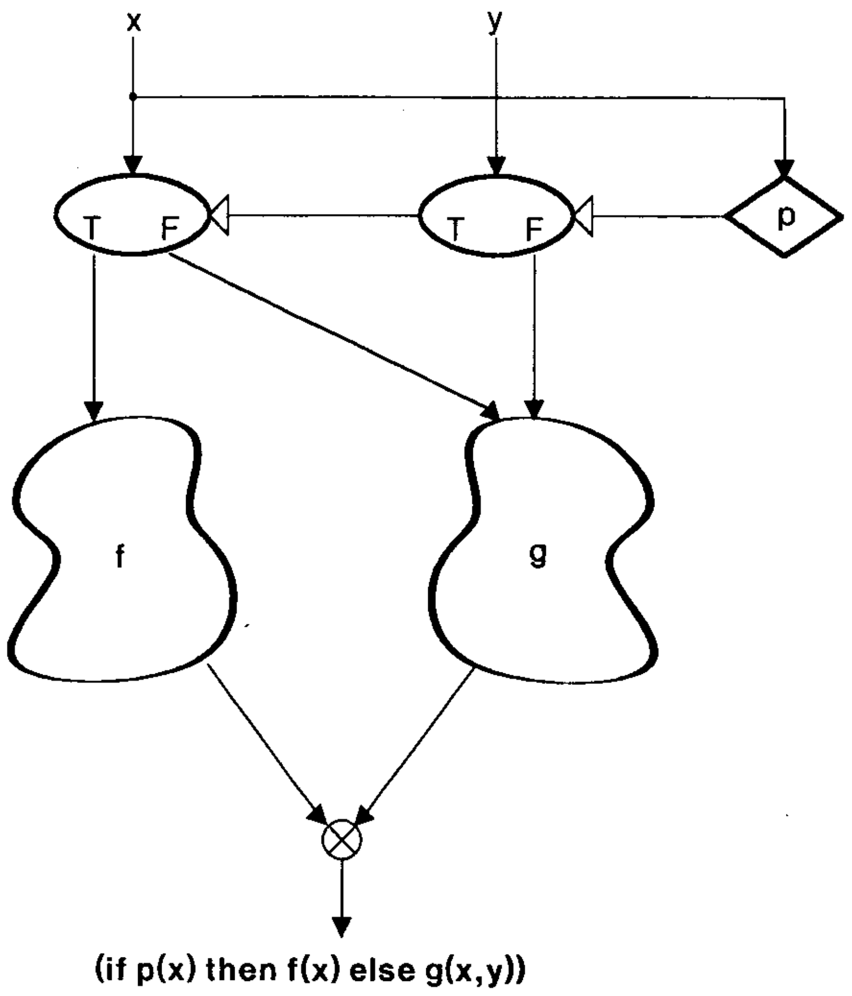

Figure 3-1: Data Flow Program Segment with Graph Translation

dependencies apparent. Nodes in the graph are called *actors*; the interconnections between actors are called *links* (see Figure 3-1). Execution of a data flow program is modeled by data values (*tokens*) flowing across the links in the form of packets.[3]

It is the execution model that distinguishes data flow. While it is true that many compilers of von Neumann style languages also use a directed-graph representation of the program at some point, the executable result is a sequentialization of the graph for the base hardware. A data flow machine, on the other hand, operates on the directed graph itself.

[3]In the VAL execution model, the only information that travels on a link from one actor to the next is value information. In the ID execution model, a tag travels with a value. The tag uniquely identifies the context in which this value is to be used.

The mechanism by which this is done is relatively straightforward: the actors in the graph are stored in the data flow machine's program memory. Associated with each actor is the memory necessary to contain the *values* of tokens which sit on its input links. In its most simplistic form, a data flow machine works by executing the instructions associated with any actors in the graph that satisfy the condition that all inputs are known. These actors will, in turn, create new tokens (on their output links) which are then distributed to other actors according to the connections in the directed graph.

In the cases of recursive applications of procedures and of program loops, it is desirable to use the same actors to represent multiple invocations (this is roughly the same as the argument in favor of reentrant code). It is necessary to provide a mechanism which will keep tokens from getting "mixed up" - one such scheme involves tagging the tokens [3, 10]. The tags are constructed in such a way that the only tokens which will have the same tag are the ones which should intentionally be paired (*e.g.*, the two input tokens for a particular ADD actor). Thus, the token carries both data and context information.

This model of computation is inherently deterministic; given the same set of inputs, a data flow graph will always produce the same result independent of issues related to concurrency, timing, etc. In this sense, data flow languages and purely functional languages do not differ. To interface a programming language to the (inherently nondeterministic) real world, the capability to perform nondeterministic computation must be introduced into these languages. One approach [3] is to encapsulate the nondeterminism with a special data flow construct called a *manager*. A single manager may be used by many fragments of the total graph; use of the manager constitutes sending tokens to the manager's input. It is at this point that nondeterminism arises - the entry to the manager merges the tokens *as they arrive*, paying no attention to any specific ordering. The manager then deals with these coalesced inputs as a stream.

3.2. Differences that Influence Debugging

Many notions of debugging that programmers have developed in their dealings with von Neumann machines have no direct interpretation in the context of data flow machines. We examine a few simple cases:

- **Breakpointing and halting of execution:** The idea of disrupting the normal control flow when execution of a program reaches a certain well-defined state is strongly at odds with the data flow model. First, there is no sense of *control flow* at all. Second, a program in execution has no readily identifiable states.

- **Examining memory:** Data flow has exorcised all forms of global memory. Hence, there is no strongly analogous concept. Journal information (described below) may be an exception.

To further complicate matters, some data flow languages, including ID, provide a facility for treating procedures as first-class citizens (*i.e.*, procedure definitions can be logically carried on tokens). The procedure is then used in a specific context by a special actor called APPLY.

Another problem is that in order to debug, the user must reproduce the bug. Reproduction of the same situation under which the bug occurred can be a difficult problem in data flow or in a von Neumann machine in nondeterministic situations. In data flow, we have restricted the nondeterminism to the nondeterministic merge actor. It is possible to keep a journal[4] of the history of each nondeterministic merge (*e.g.*, a time-ordered log of its outputs). The journal will allow a historical reconstruction of all the data flow tokens that have passed through the nondeterministic merge. A linguistic extension is necessary to make use of journal data.

There are two possible definitions of error values in programming languages [11]:

1. "Error values may be defined as particular values within a larger type domain, and each type domain contains distinct error values; or

2. "Error values are defined as members of a distinct type domain."

where *larger type domain* denotes INTEGER, REAL, BOOLEAN, etc. and *distinct type domain* denotes the class of all errors. An example of the first alternative is an integer overflow or a numeric zero-divide. This corresponds to the strong typing of VAL. In the second alternative, a zero-divide is simply an error type. This corresponds to the lack of strong typing in ID.

Errors are also handled differently in data flow than in conventional languages in that occurrence of error values does not stop execution. Since there is no global control, it is unclear what stopping the computation would involve, or even that stopping the computation would be meaningful. Further, if the value of the error is not related to the computation's answer, there is no reason to stop the computation. In the present implementations, the error value is propagated through each operation where an error value was input. No attempt is made to keep track of where the error first occurred.

3.3. Debugging Existing Functional Languages

One of the earliest operational functional languages was the *pure LISP* subset of LISP 1.5. Its debugging environment [9] was characterized by

- **Explicit error messages:** These were generated at compile time (for the LISP compiler), assembly time (for LAP - the LISP Assembly Program), and run time (interpreter, garbage collector, etc.).

- **TRACE facility:** Similar to the LISP TRACE function of today; it was useful in displaying the behavior of recursive functions. It would print both the function name and its arguments upon function entry and exit.

- **OVERLORD:** A simple run time monitor. Its primary user inputs came from switches

[4]The authors are grateful to William Ackerman for this idea.

and buttons on the machine's console. As stated in the LISP 1.5 manual [9], "It controls the handling of tapes, the reading and writing of entire core images, the historical memory of the system, and the taking of dumps." This is hardly in line with the goal of allowing debugging at the level of the language in which the program is written.

- **Miscellaneous debug support:** Other functions are provided for debugging: COUNT (for breaking out of program loops; it generates an error after a user-specified number of CONSes are created); ERRORSET (for user-handling of errors); TRACECOUNT (allows tracing to be enabled *after* a fixed number of function entrances - to limit useless TRACE printout); and BACKTRACE (to control the backtrace display if an error should occur).

A more recent system is the Experimental Formal Functional Programming system (FPX) of Thomas [13]. It allows debugging at the language level (FFP), but the debugging and environmental commands are metalinguistic rather than being part of the language itself. Simple tracing (display of an application prior to evaluation on a named-function basis) is permitted, and works with user-defined functions as well as the predefined FFP forms (*e.g., COND, INSERT, ALPHA*). Unfortunately, FPX is not a *modeless* system; debug mode is separate from non-debug mode.

ID and VAL have simple debuggers as well. While both languages have sequential simulators (with associated debuggers), neither has a defined debugging methodology on a data flow machine. The debuggers are built into

- **IDSYS:** A compiler, simulator, and debugger for the Irvine Dataflow language which supports tracing of non token-carried procedures by name.

- **VALSYS:** A simulator and debugger for the Value-Oriented Algorithmic Language which supports tracing of procedures by name. Values of actual parameters can be displayed upon procedure entry and/or exit. Classes of commands like all operations, all errors, and all iterations can be traced also. Breakpoints occur when a traced item is found. At the breakpoint, identifiers that are in the current environment or that are in an active call of a procedure can be examined [5].

4. Goals of a Data Flow Debugging Device

It is important to determine the goals of the data flow debugging device before making any proposals. Although many of these seem interrelated, and the level of specificity is not consistent, each of them was considered in making our proposals.

- **Give the user the information he wants.** The programmer should not have to wade through mounds of meaningless data to find information he is looking for. On the other hand, he should not have to struggle to get enough data. The debugger should be flexible enough to allow the programmer to get as much or as little information as he needs. This implies that all information possible should be available to the programmer, but he shouldn't need to look at it all.

- **Allow the user to interact with the debugging device in concepts (and the language) with which he is familiar.** Learning a different language to communicate with the debugging device is very inconvenient and, we believe, unnecessary. Extensions to the existing languages should be minimal and will allow the user to easily learn to debug his programs.

- **Impose no debugging modes on the user.** Debugging should be an extension of the interactive environment. Debugging strategies should not be available only when in a debugging mode. The debugging commands may be valuable for use in many circumstances where the mode requirement would rule out their use[5].

- **Preserve the circumstances in which the bug occurred.** Although the debugging device itself may not be able to guarantee this, it is important that the device not violate this goal. The bug must be duplicated while using the debugging device to most easily debug the program.

- **Preserve the semantic base of the applicative system, even when debugging [11.]** This implies that activities must be considered to happen asynchronously, and the debugging device should not change this view.

- **Assist the user in finding bugs in his program.** This involves human factors issues and an understanding of the debugging process. All previous goals can be considered to be subgoals to this goal. Rather than lose sight of the overall objective, it seemed safest to list this, along with its subparts.

5. Proposals

This section details ideas for a good data flow debugging methodology. We have restricted ourselves to the cases wherein the specifications, algorithms, and encodings may be in error; we assume that the language translation process, the hardware, and the bug detection mechanism are all operating properly.

We propose a total, interactive environment as the preferred embodiment of the debugging tools for data flow programs. This section describes the individual components of the environment (as they relate to debugging). We conclude with a discussion of the total environment.

[5]The idea for lack of modes is partly based on [12].

5.1. Watching Execution

In sequential debugging, it is often valuable to *single step* through the program, or a portion of it. In data flow, it might be desirable to simulate execution for debugging purposes. The user should have control over what is displayed and the rate at which the display occurs.[6]

We are very attracted by the window mechanism in Smalltalk [12], and envision a similar mechanism for use in our debugger. Graph-like flow on a statement-by-statement basis would then be shown. Parallel execution of procedures could occur on several windows on the screen, headed by information about what context the statement was being executed in. The user could select which window he wished to examine at any given point. Because of the large amount of parallelism that may be present, it might become necessary during execution to further restrict the amount of information the user can see at one time to less than one procedure.

In addition to being a debugging aid, this could prove to be a conceptual aid for people who are accustomed to imperative, sequential languages. Watching execution of a data flow program could aid in understanding the reasons for the difference between a data flow language and an imperative language.

Much more development will need to be done on this approach to examine its desirability, feasibility, and implementation.

5.2. Simple Monitoring

As we have seen, it is frequently sufficient to allow the programmer the ability to "look" at values as they are created. Three simple techniques along these lines are described.

5.2.1. Asynchronous Monitors

Due to the single assignment principle, variable names in data flow programs denote the output of a single actor in the data flow graph. All links emerging from this actor convey tokens associated with the named variable in the data flow program. Hence, by adding an extra link to any actor (and logically connecting this link to an output device), the programmer could monitor the values of tokens being created by the actor without disrupting the normal operation of the graph (Figure 5-1). It will be necessary to associate uniquely specified variable names with the actors in the program graph to facilitate this. This information is available from the compiler's symbol table.

5.2.2. Input-Influence tracing

To aid in problem isolation, it may be convenient to identify which, if any, input values influence certain output values. We may view the technique of tracing the ancestry of a token in an inverse way; *i.e.*, "given an input value, which outputs does it influence?" This kind of value tracing is simply effected by marking the chosen input token in a special way (*i.e.*, make it "radioactive"). If the semantics of token propagation are changed slightly (for any actor, its output will be radioactive

[6]Information would be displayed regarding any statements that had been optimized out by the compiler.

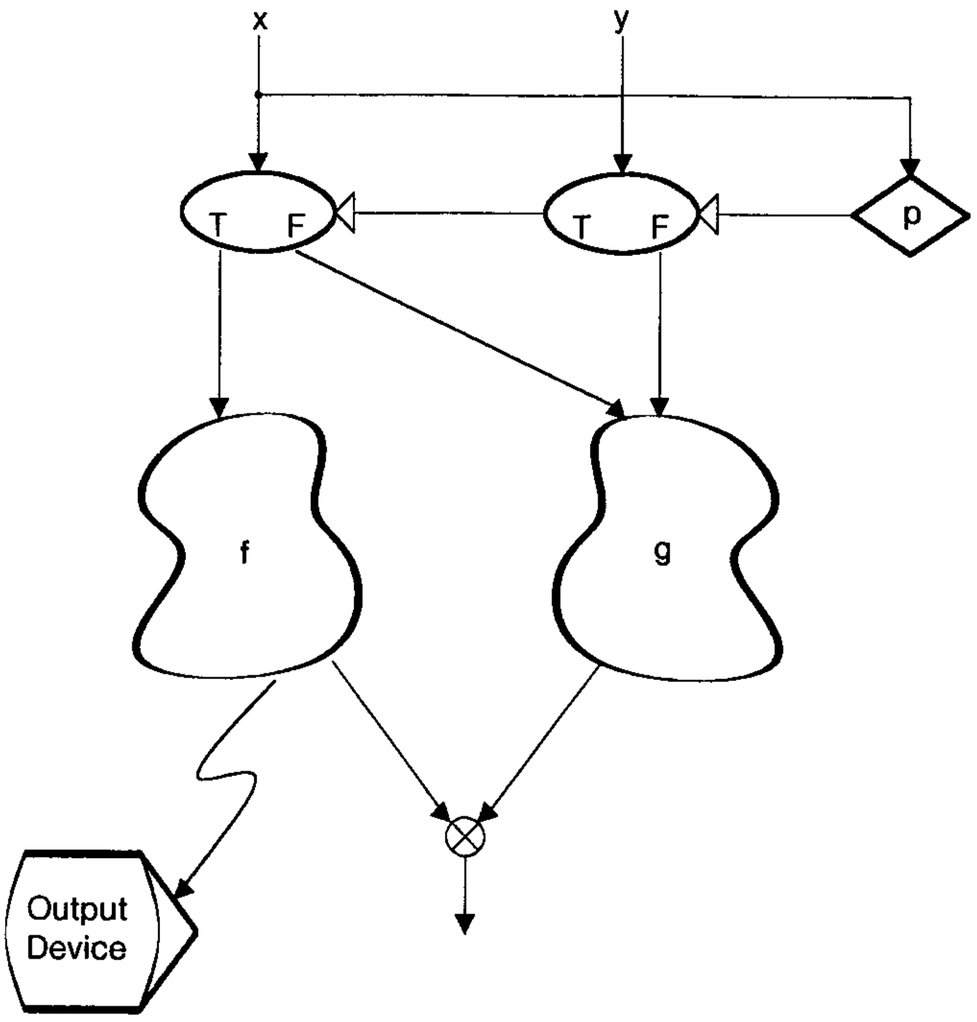

Figure 5-1: Asynchronous Monitor

if any input is radioactive), the input token will generate a radioactive history which can be sampled and monitored using the other techniques presented here.

5.2.3. Error tracing

Because of the error-propagation method of error handling, it is a very complex task to discover where an error first occurred. It is, therefore, important to discover through the debugging device when an error value is generated (*e.g.*, integer underflow, integer overflow, etc.) for the first time, and possibly to trace its progress. This could be done by examining the values on links in the graph associated with variable names.

As an extension to this idea, specified errors could be traced. The user could select tracing of all error values that occur, or tracing of only specific error values.

5.3. Intelligent monitoring

Unfortunately, while such simple techniques can be helpful in restricted cases, they can also create lots of useless information (*viz.* the LISP TRACECOUNT function) in the process. Hence, we propose some slightly more sophisticated techniques for extracting only that information that the programmer deems as "useful", to wit, daemons and linguistic extensions.

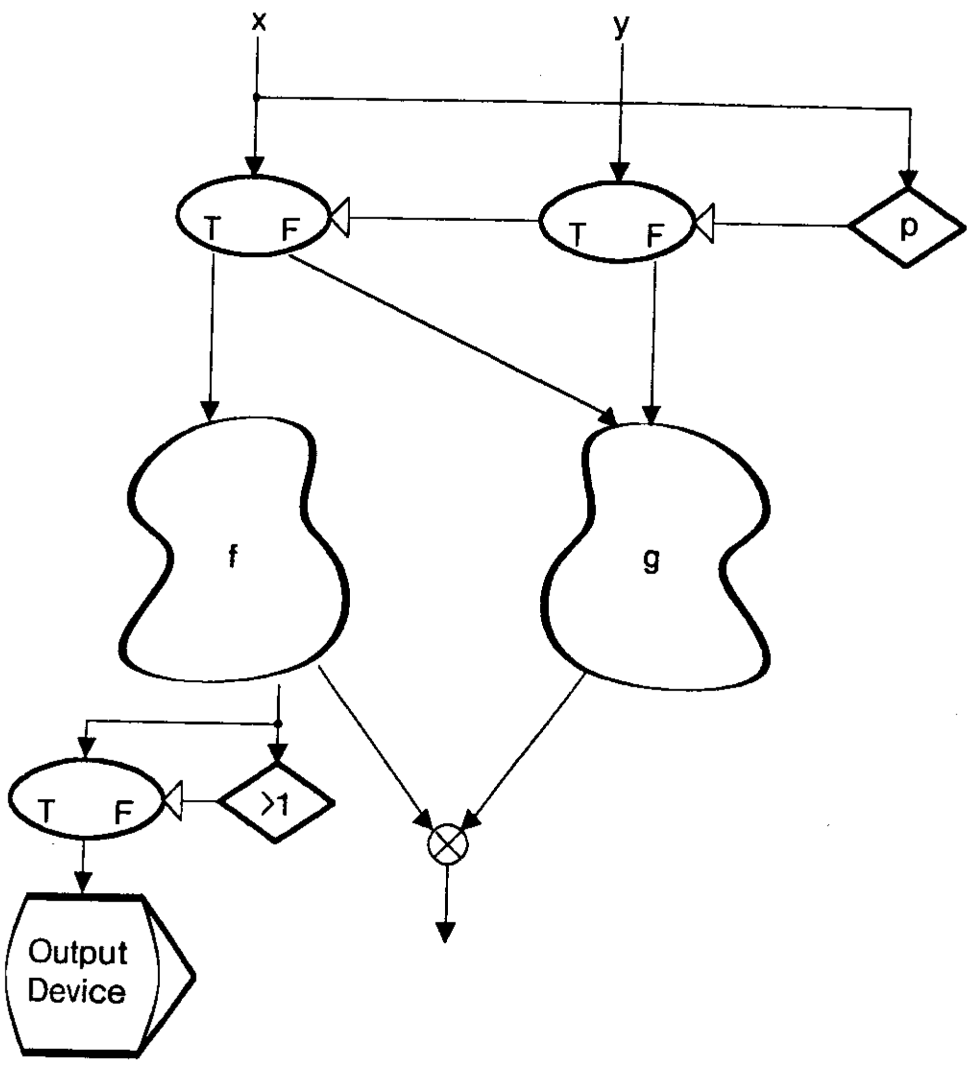

Figure 5-2: Daemon

5.3.1. Daemons

To extend the concept of simple, asynchronous monitoring, it should also be possible to allow the user to create an arbitrary program (compiled into a directed graph, of course) which is dynamically linked into another graph in much the same fashion as the simple monitor. If the user wishes to look at all executions of a loop, an error, or a procedure, that information should be easily accessed. If, however, he wishes to look at only a particular execution, that flexibility should be available to him. Since this "debug" program is written in the same language as the program being debugged, the user has the full flexibility of the language in sifting through the sampled tokens to extract those which are of interest (*e.g.*, looking only at the values of loop variables on the 1000th iteration but ignoring all others). This will permit not only the identification of loop-instances and error-instances, but also the identification of procedure applications at arbitrary nesting levels. Figure 5-2 demonstrates the attachment of a very simplistic daemon which is acting as a filter (*i.e.*, only values that are less than 1 will be passed to the output device).

It may also be valuable to allow values of these variables to be changed, and to continue execution of the program with the variable's new value. This would require that the original value not be sent to the next operation. This could be done by redirecting the output of the actor creating the value.

5.3.2. Linguistic Extensions

A mechanism like CLU's exception handler [7] is desirable to allow the programmer to deal with errors as they arise (in a hierarchical fashion). While not explicitly a part of a debugger, it is an inherent part of the total bug-reduction process. Such an exception handler was proposed by Plouffe [11] for a value-based applicative semantic model. Although his exception handler has many desirable traits, handling an exception may impose both sequencing constraints and processing overhead. The claim is that such overhead is unavoidable in a distributed processing environment.

The daemons mentioned above will require careful extension of the language to allow the programmer to deal with context-identifying information. In VAL, this can be done by carrying this information as a part of the argument list or environment that is available to operations. In ID, this information is already carried as a part of the token (the *⟨u.c.s.i.⟩* tag of [3]).

5.4. Combined Program Development, Execution and Debugging

It is our contention that any general purpose computer system (data flow or otherwise) should be capable of interacting with a human user in a manner that can be considered as an extension to his thought processes. No fictitious boundaries or limits should be visible [12]. To this end, we have developed our notions of a data flow debugger with a strong adherence to elimination of modes. Further, we believe that all tasks that involve the creation, encoding, translation, execution, and debugging of a program should be consolidated into a single environment (*e.g.*, debugging tools should be built into the language rather than being separate from it).

Figure 5-3 is a model for a data flow debugger. Although the concepts used are oriented toward ID, they are only meant to be suggestive of a method, not an implementation. The interpretation of the model is that the user himself becomes a part of the directed graph; his nondeterminate behavior is encapsulated as a manager.

Referring to the figure, we see mechanisms provided for the gathering of tokens from various parts of the graph (used for asynchronous monitoring) and the ability to return values from the

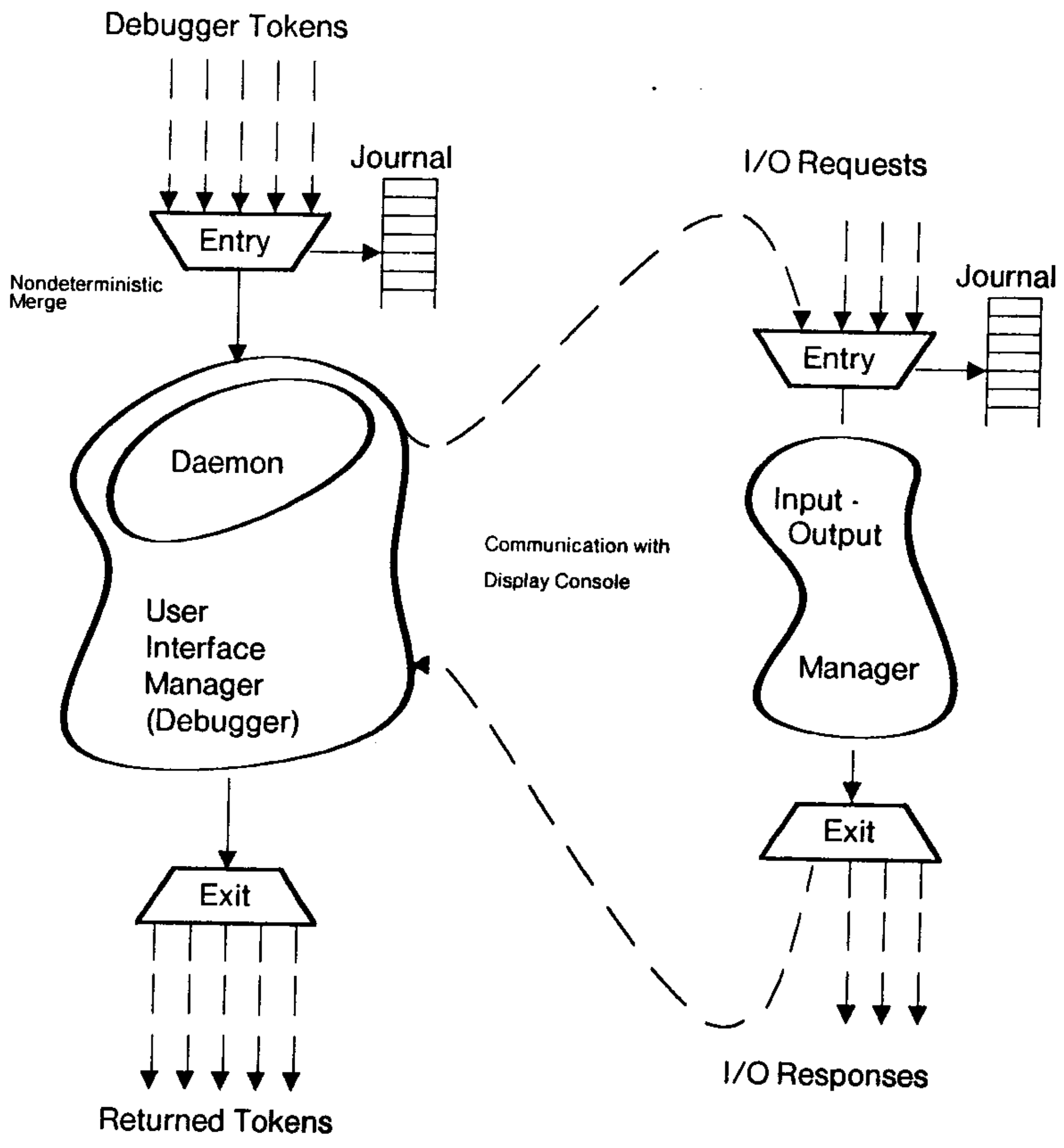

Figure 5-3: Implementation of a Debugger as an ID Manager

manager (as is necessary when value modification is desired). The user-written parts of this manager can be used for daemon definition (intelligent monitoring). Functions like display screen manipulation (SMALLTALK-like windows, etc.) can also be performed wholly within the language. This manager will, in turn, communicate with yet another manager to perform the actual input/output. It should be understood that this user interface has the responsibility for mediating *all* communication with the user (not just for debugging); editing, compiling, and other related functions are all provided by this manager.

6. Conclusion

Interactive debugging environments are still being addressed for von Neumann-dependent programming languages. In the course of researching this paper, it became apparent that little information is published on debugging environments. Only a paragraph here, a sentence there, could be gleaned from publications on programming languages and programming environments. We drew largely on our experience as programmers (and debuggers) to determine what information would be useful to the user of a data flow debugging device. We believe that the methods we have proposed will be valuable for data flow debugging, but little experience has yet been gained in debugging a data flow program on a data flow machine. Further experience with data flow languages on data flow machines is necessary.

We have proposed a general implementation scheme for our proposals, but there are many detailed implementation questions yet to be answered. Other questions about how much these schemes will hinder efficiency have yet to be addressed. We did not abandon powerful tools on efficiency grounds, although we did try to weed out the most impractical schemes.

Since it is early in the history of data flow and since debugging issues that arise in this context are only beginning to be addressed, experience will best answer many of the questions this paper raises.

References

1. Ackerman, W. B. Data Flow Languages. In *AFIPS Conference Proceedings, Volume 48: Proceedings of the 1979 National Computer Conference,* AFIPS. 1979, pp. 1087-1095.

2. Ackerman, W. B. and Dennis, J. B. VAL -- A Value - Oriented Algorithmic Language: Preliminary Reference Manual. Tech. Rep. 218, Laboratory for Computer Science, MIT, Cambridge, Mass., December, 1978.

3. Arvind, K. P. Gostelow, and W. Plouffe. An Asynchronous Programming Language and Computing Machine. Tech. Rep. 114a, Department of Information and Computer Science, University of Californiav, Irvine, California, December, 1978.

4. Backus, J. Can Programming Be Liberated from the von Neumann Style? A Functional Style and Its Algebra of Programs. *Comm. ACM 21,* 8 (August 1978), 613-641.

5. Brock, J. D. Val Big Doc 2. Unpublished, MIT, 1981

6. Dennis, J. B. First Version of a Data Flow Procedure Language. In *Lecture Notes in Computer Science, Volume 19: Programming Symposium: Proceedings, Colloque sur la Programmation,* B. Robinet, Ed., Springer-Verlag, 1974, pp. 362-376.

7. Liskov, Barbara, et. al. CLU Reference Manual. Tech. Rep. TR-225, Laboratory for Computer Science, MIT, Cambridge, Mass., October, 1979.

8. McCarthy, J. Recursive functions of symbolic expressions and their computation by machine, Part 1. *Comm. ACM 3,* 4 (April 1960), 184-195.

9. McCarthy, J., Abrahams, P. W., Edwards, D. J., Hart, T. P. and Levin, M. I. *LISP 1.5 Programmer's Manual.* MIT Press, Cambridge, Mass., 1965.

10. Miranker, G.S. Implementation Schemes for Data Flow Procedures. Memo 138-1, Computation Structures Group, Laboratory for Computer Science, MIT, Cambridge, Mass., May, 1976.

11. Plouffe, W.E. Exception Handling and Recovery in a Dataflow System. Ph.D. Th., Department of Information and Computer Science, University of Californiav, Irvine, California, 1979.

12. Tesler, L. The Smalltalk Environment. *BYTE* (August 1981), 90-145.

13. Thomas, R.E. The Experimental Formal Functional Programming Language Interpreter (FPX). Massachusetts Institute of Technology, Cambridge, Mass., April, 1981.