Advances in Dataflow Programming Languages

WESLEY M. JOHNSTON, J. R. PAUL HANNA, AND RICHARD J. MILLAR

University of Ulster

Abstract. Many developments have taken place within dataflow programming languages in the past decade. In particular, there has been a great deal of activity and advancement in the field of dataflow visual programming languages. The motivation for this article is to review the content of these recent developments and how they came about. It is supported by an initial review of dataflow programming in the 1970s and 1980s that led to current topics of research. It then discusses how dataflow programming evolved toward a hybrid von Neumann dataflow formulation, and adopted a more coarse-grained approach. Recent trends toward dataflow visual programming languages are then discussed with reference to key graphical dataflow languages and their development environments. Finally, the article details four key open topics in dataflow programming languages.

Categories and Subject Descriptors: A.1 [**Introductory and Survey**]; C.1 [**Processor Architectures**]; D.2 [**Software Engineering**]; D.3 [**Programming Languages**]

General Terms: Languages, Theory

Additional Key Words and Phrases: Dataflow, software engineering, graphical programming, component software, multithreading, co-ordination languages, data flow visual programming

1. INTRODUCTION

The original motivation for research into dataflow was the exploitation of massive parallelism. Therefore, much work was done to develop ways to program parallel processors. However, one school of thought held that conventional "von Neumann" processors were inherently unsuitable for the exploitation of parallelism [Dennis and Misunas 1975; Weng 1975]. The two major criticisms that were leveled at von Neumann hardware were directed at its global program counter and global updatable memory [Silc et al. 1998], both of which had become bottlenecks [Ackerman 1982; Backus 1978]. The alternative proposal was the dataflow architecture [Davis 1978; Dennis and Misunas 1975; Weng 1975], which avoids both of these bottlenecks by using only local memory and by executing instructions as soon as their operands become available. The name dataflow comes from the conceptual notion that a program in a dataflow computer is a directed graph and that data flows between instructions, along its arcs [Arvind and Culler 1986; Davis and Keller 1982; Dennis 1974; Dennis and Misunas 1975]. Dataflow hardware architectures

Authors' addresses: Faculty of Engineering, University of Ulster, Newtownabbey, Northern Ireland, BT37 0QB; email: W. M. Johnston, wesley@wesleyjohnston.com; J. R. P. Hanna and R. J. Millar, {p.hanna,rj.millar}@ulster.ac.uk.

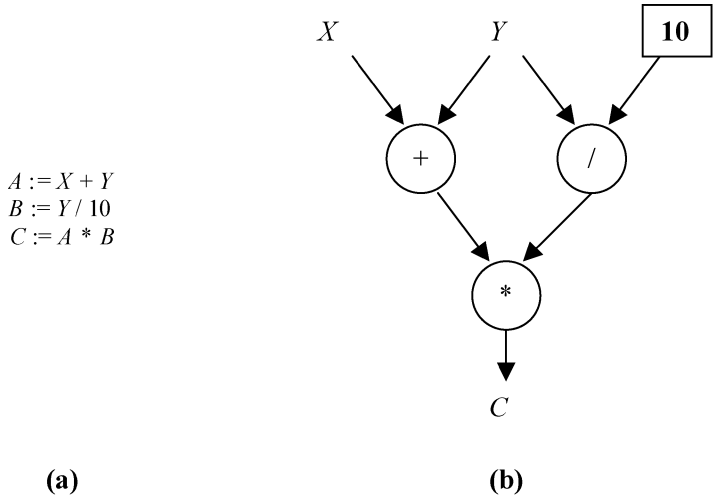
looked promising [Arvind and Culler 1986; Dennis 1980; Treleaven and Lima 1984; Veen 1986], and a number of physical implementations were constructed and studied (for examples, see Davis [1978], Keller [1985], Papadopoulos [1988], Sakai et al. [1989], and Treleaven et al. [1982]).

Faced with hardware advances, researchers found problems in compiling conventional imperative programming languages to run on dataflow hardware, particularly those associated with side effects and locality [Ackerman 1982; Arvind et al. 1977; Arvind and Culler 1986; Kosinski 1973; Wail and Abramson 1995; Weng 1975; Whiting and Pascoe 1994]. They found that by restricting certain aspects of these languages, such as assignments, they could create languages [Ackerman 1982; Ashcroft and Wadge 1977; Dennis 1974; Hankin and Glaser 1981; Kosinski 1978] that more naturally fitted the dataflow architecture and could thus run much more efficiently on it. These are the so-called dataflow programming languages [Ackerman 1982; Whiting and Pascoe 1994] that developed distinct properties and programming styles as a consequence of the fact that they were compiled into dataflow graphs—the "machine language" of dataflow computers.

The often-expressed view in the 1970s and early 1980s that this form of dataflow architecture would take over from von Neumann concepts [Arvind et al. 1977; Treleaven et al. 1982; Treleaven and Lima 1984] never materialized [Veen 1986]. It was realized that the parallelism used in dataflow architectures operated at too fine a grain and that better performance could be obtained through hybrid von Neumann dataflow architectures. Many of these architectures [Bic 1990] took advantage of more coarse-grained parallelism where a number of dataflow instructions were grouped and executed in sequence. These sets of instructions are, nevertheless, executed under the rules of the dataflow execution model and thus retain all the benefits of that approach. Most dataflow architecture efforts being pursued today are a form of hybrid [Iannucci 1988; Nikhil and Arvind 1989], although not all, for example, Verdoscia and Vaccaro [1998].

The 1990s saw a growth in the field of dataflow visual programming languages (DFVPLs) [Auguston and Delgado 1997; Baroth and Hartsough 1995; Bernini and Mosconi 1994; Ghittori et al. 1998; Green and Petre 1996; Harvey and Morris 1993, 1996; Hils 1992; Iwata and Terada 1995; Morrison 1994; Mosconi and Porta 2000; Serot et al. 1995; Shizuki et al. 2000; Shürr 1997; Whiting and Pascoe 1994; Whitley 1997]. Some of these, such as LabView and Prograph were primarily driven by industry, and the former has become a successful commercial product that is still used today. Other languages, such as NL [Harvey and Morris 1996], were created for research. All have software engineering as their primary motivation, whereas dataflow programming was traditionally concerned with the exploitation of parallelism. The latter remains an important consideration, but many DFVPLs are no longer primarily concerned with it. Experience has shown that many key advantages of DFVPLs lie with the software development lifecycle [Baroth and Hartsough 1995].

This article traces the development of dataflow programming through to the present. It begins with a discussion of the dataflow execution model, including a brief overview of dataflow hardware. Insofar as this research led to the development of dataflow programming languages, a brief historical analysis of these is presented. The features that define traditional, textual dataflow languages are discussed, along with examples of languages in this category. The more recent trend toward large-grained dataflow is presented next. Developments in the field of dataflow programming languages in the 1990s are then discussed, with an emphasis on DFVPLs. As the environment is key to the success of a DFVPL, a discussion of the issues involved in development environments is also presented, after which four examples of open issues in dataflow programming are presented.



```
A := X + Y
B := Y / 10
C := A * B
```

Fig. 1. A simple program (a) and its dataflow equivalent (b).

2. THE DATAFLOW EXECUTION MODEL

2.1. The Pure Dataflow Model

In the dataflow execution model, a program is represented by a directed graph [Arvind and Culler 1986; Davis and Keller 1982; Dennis 1974; Dennis and Misunas 1975; Karp and Miller 1966]. The nodes of the graph are primitive instructions such as arithmetic or comparison operations. Directed arcs between the nodes represent the data dependencies between the instructions [Kosinski 1973]. Conceptually, data flows as tokens along the arcs [Dennis 1974] which behave like unbounded first-in, first-out (FIFO) queues [Kahn 1974]. Arcs that flow toward a node are said to be *input* arcs to that node, while those that flow away are said to be *output* arcs from that node.

When the program begins, special activation nodes place data onto certain key input arcs, triggering the rest of the program. Whenever a specific set of input arcs of a node (called a *firing* set) has data on it, the node is said to be *fireable* [Arvind and Culler 1986; Comte et al. 1978; Davis and Keller 1982]. A fireable node is executed at some undefined time after it becomes fireable. The result is that it removes a data token from each node in the firing set, performs its operation, and places a new data token on some or all of its output arcs. It then ceases execution and waits to become fireable again. By this method, instructions are scheduled for execution as soon as their operands become available. This stands in contrast to the von Neumann execution model, in which an instruction is only executed when the program counter reaches it, regardless of whether or not it can be executed earlier than this.

The key advantage is that, in dataflow, more than one instruction can be executed at once. Thus, if several instructions become fireable at the same time, they can be executed in parallel. This simple principle provides the potential for massive parallel execution at the instruction level.

An example of dataflow versus a traditional sequential program is shown in Figure 1. Figure 1(a) shows a fragment of program code and Figure 1(b) shows how this is represented as a dataflow graph. The arrows represent arcs, and the circles represent instruction nodes. The square represents a constant value, hard-coded into the program. The letters represent where data flows in or out of the rest of the program, which is not shown. Where more than one arrow emanates from a given input, it means that the single value is duplicated and transmitted down each path.

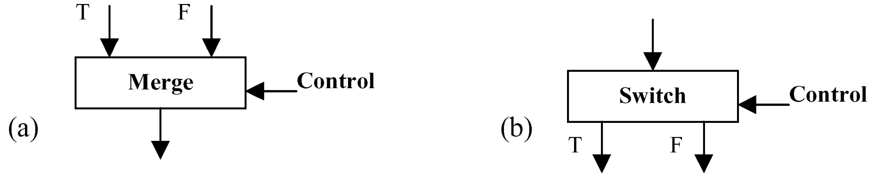


Fig. 2. Gates in a dataflow graph.

Under the von Neumann execution model, the program in Figure 1(a) would execute sequentially in three time units. In time unit 1, X and Y are added and assigned to A. In time unit 2, Y is divided by 10 and assigned to B. In time unit 3, A and B are multiplied together and assigned to C.

Under the dataflow execution model, where the graph in Figure 1(b) is the machine code, the addition and division are both immediately fireable, as all of their data is initially present. In time unit 1, X and Y are added in parallel with Y being divided by 10. The results are placed on the output arcs, representing variables A and B. In time unit 2, the multiplication node becomes fireable and is executed, placing the result on the arc representing the variable C. (In dataflow, every arc can be said to represent a variable.) In this scenario, execution takes only two time units under a parallel execution model.

It is clear that dataflow provides the potential to provide a substantial speed improvement by utilizing data dependencies to locate parallelism. In addition, if the computation is to be performed on more than one set of data, the calculations on the second wave of values of X and Y can be commenced before those on the first set have been completed. This is known as *pipelined* dataflow [Gao and Paraskevas 1989; Wadge and Ashcroft 1985] and can utilize a substantial degree of parallelism, particularly in loops, although techniques exist to utilize greater parallelism in loops [Arvind and Nikhil 1990]. A dataflow graph that produces a single set of output tokens for each single set of input tokens is said to be *well-behaved* [Dennis 1974; Weng 1975].

Another key point is that the operation of each node is functional. This is because data is never modified (new data tokens are created whenever a node fires), no node has any side effects, and the absence of a global data store means that there is locality of effect. As a result of being functional, and the fact that the data travels in ordered queues, a program expressed in the pure dataflow model is determinate [Arvind and Culler 1986; Davis and Keller 1982; Kahn 1974]. This means that, for a given set of inputs, a program will always produce the same set of outputs. This can be an important property in certain applications. Some research has been done on the implications of nondeterminate behavior [Arvind et al. 1977; Kosinski 1978] and this is discussed further in Section 6.4.

2.1.1. Controlling Data Tokens. In Figure 1(b) the two arcs emanating from input Y signify that that value is to be duplicated. Forking arcs in this manner is essential if a data token is needed by two different nodes. A data token that reaches a forked arc gets duplicated and a copy sent down each branch. This preserves the data independence and the functionality of the system.

To preserve the determinacy of the token-flow model, it is not permitted to arbitrarily merge two arcs of flowing data tokens. If this were allowed, data could arrive at a node out of order and jeopardize the computation. It is obvious, however, that it would be difficult indeed to express a program in the dataflow model if arcs could only be split and never merged. Thus, the dataflow model provides special control nodes called *gates* [Davis and Keller 1982; Dennis 1974] that allow this to happen within well-controlled limits. Figure 2(a) shows a *Merge* gate. This gates takes two data input arcs, labeled the *true* and *false* inputs, as well as a "control" input arc that carries a Boolean value. When the node fires, the control token is absorbed first. If the value is true, the token on the true input is absorbed and placed on the output. If it is false, then the token on the false input is absorbed and placed on the output. Figure 2(b) shows a *Switch* gate. This gate operates in much the same way, except that there is a single input and the control token determines on which of two outputs it is placed.

A full treatment of controlling tokens to provide conditional and iterative execution is given in Section 6.2. At this stage, it is sufficient to say that by grouping together three Switch gates, it is possible to implement well-behaved conditional execution, while the combined use of both types of gate implements well-behaved iterative execution.

2.1.2. An Alternative to Token-Based Dataflow. The pure dataflow execution model outlined above is based on flowing data tokens, like most dataflow models. However, it should be pointed out that an alternative, known as the *structure model*, has been proposed in the literature. Expounded by Davis and Keller [1982] and Keller and Yen [1981], it contains the same arc-and-node format as the token model. In the structure model, however, each node creates only one data object on each arc that remains there: the node builds one or more data structures on its output arcs. It is possible for these structures to hold infinite arrays of values, permitting open-ended execution, and creating the same effect as the token model, but with the advantage that the structure model permits random access of the data structures and history sensitivity.

The key difference between the structure model and the token model is the way they view data. In the token model, nodes are designed to be stream processors, operating on sequences of related data tokens. In the structure model, however, the nodes operate on structures and have no concept of streams of data structures. As a consequence, a structure model will need a more complex supporting language [Davis and Keller 1982].

Initially, the structure model seems attractive. Token streams can be represented by infinite objects with the advantage that the streams can be accessed randomly and that the entire history of a stream can be accessed without needing to explicitly preserve earlier data from the stream. Additionally, the point is made that token models force the programmer to model all programs as token streams, while the structure model allows them to make the choice [Davis and Keller 1982].

However, the structure model has the key disadvantage that it cannot store data efficiently. It requires all data generated to be stored, to preserve the ability to examine the history of the program. Token models are inherently more efficient at storing data. Some of this problem can be alleviated by compiler efficiency, but it is a complex process. Despite research in the area in the early 1980s [Davis and Keller 1982; Keller and Yen 1981], the structure model was not widely adopted into dataflow, which remains almost exclusively token-based.

2.1.3. Theoretical Implementation: Data and Demand-Driven Architectures. The earliest dataflow proposals imagined data tokens to be passive elements that remained on arcs until they were read, rather than actually controlling the execution [Kosinski 1973]. However, it quickly became normal in dataflow projects for data to control the execution. There were two ways of doing this in a theoretical implementation of the pure dataflow model.

The first approach is known as the *data-driven approach* [Davis and Lowder 1981; Davis and Keller 1982; Dennis 1974; Treleaven et al. 1982], although this term is slightly misleading as both approaches can be said to be driven by data, insofar as they follow the principles of dataflow. This approach should really be termed the data-availability-driven approach because execution is dependent on the availability of data. Essentially a node is inactive while data is arriving at its inputs. It is the responsibility of an overall management device to notify and fire nodes when their data has arrived. The data-driven approach is a two-phase process where (a) a node is activated when its inputs are available, and (b) absorbs its inputs and places tokens on its output arcs.

The second approach is the *demand-driven approach* [Davis and Keller 1982; Kahn 1974]. In this approach, a node is activated only when it receives a request for data from its output arcs. At this point, it demands data from all relevant input arcs. Once it has received its data, it executes and places data tokens on its output arcs. The demand-driven approach is thus a four-phase process [Davis and Keller 1982] where (a) a node's environment requests data, (b) the node is activated and requests data from its environment, (c) the environment responds with data, and (d) the node places tokens on its output arcs. Execution of the program begins when the graph's environment demands some output from the graph.

Each of these approaches has certain advantages. The data-driven approach has the advantage that it does not have the extra overhead of propagating data requests up the dataflow graph. On the other hand, the demand-driven approach has the advantage that certain types of node can be eliminated, as pointed out by Davis and Keller [1982]. This is because only needed data is ever demanded. For example, the Switch, node, shown in Figure 2(b), is not required under a demand-driven approach because only one of the True or False outputs will demand the input, but not both. Therefore, they can both be attached directly to the input. This is one example of how programming with dataflow can be affected by the choice of physical implementation, or at least by the choice of execution model.

It can also be argued that the demand-driven approach prevents the creation of certain types of programs. For example, modern software is often event-driven, such as for business software or real-time systems. It is not enough for the output environment to simply demand input. These examples seem to require a data-driven approach.

2.2. Early Dataflow Hardware Architectures

While dataflow seems good in theory, the practical implementation of the pure dataflow model has been found to be an arduous task [Silc et al. 1998]. There are a number of reasons for this, primarily the fact that the pure model makes assumptions that cannot be replicated in the real world. First, it assumes that the arcs are FIFO queues of unbounded capacity, but creating an unbounded memory is impossible in a practical sense. Thus any dataflow implementation is heavily tied to token-storage techniques. Second, it assumes that any number of instructions can be executed in parallel, while in reality the number of processing elements will be finite. These restrictions mean that no hardware implementation of the dataflow model will exactly mirror the pure model. Indeed, this fact can make subtle but important changes to the pure dataflow model that mean that the implementation may deadlock in cases where the pure model predicts no deadlock [Arvind and Culler 1986]. It is useful to summarize the early development of dataflow hardware in order to reinforce this point.

2.2.1. The Static Architecture. When the construction of dataflow computers began in the 1970s, two different approaches to solving the previously mentioned problems were researched. The static architecture was proposed by Dennis and Misunas [1975]. Under this architecture [Dennis 1974, 1980; Silc et al. 1998], the FIFO design of arcs is replaced by a simpler design where each arc can hold, at most, one data token. The firing rule for a node is, therefore, that a token be present on each input arc, and that there be no tokens present on any of the output arcs. In order to implement this, acknowledge arcs are implicitly added to the dataflow graph that go in the opposite direction to each existing arc and carry an acknowledgment token.

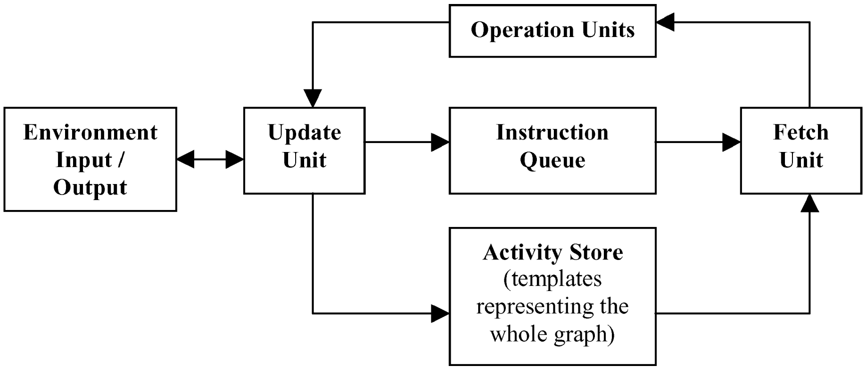

Operation Units
Environment Input / Output
Update Unit
Instruction Queue
Fetch Unit
Activity Store (templates representing the whole graph)

Fig. 3. The static dataflow architecture (based on Arvind and Culler [1986]).

The static architecture's main strength is that it is very simple and quick to detect whether or not a node is fireable. Additionally, it means that memory can be allocated for each arc at compile-time as each arc will only ever hold 0 or 1 data token. This implies that there is no need to create complex hardware for managing queues of data tokens: each arc can be assigned to a particular piece of memory store.

The graph itself is stored in the computer as a series of *templates*, each representing a node of the graph. The template holds an opcode for the node; a memory space to hold the value of the data token on each input arc, with a presence flag for each one; and a list of destination addresses for the output tokens. Each template that is fireable (the presence flag for each input is set, and that of each output is not set) has its address placed in an *instruction queue*. A *fetch unit* then repeatedly removes each template from this queue and sends an operation packet to the appropriate *operation unit*. Meanwhile, the template is cleared to prepare it for the next set of data tokens. The result is sent from the operation unit to an *update unit* that places the results onto the correct receiving arcs by reading the target addresses in the template. It then checks each template to see if it is fireable and, if so, places it in the instruction queue to complete the cycle. This process is shown in Figure 3.

Unfortunately the static model has some serious problems. The additional acknowledgment arcs increase data traffic in the system, without benefiting the computation. According to Arvind and Culler [1986], traffic can increase by a factor of 1.5 to 2.0. Because a node must wait for acknowledgment tokens to arrive before it can execute again, the time between successive firings of a node increases. This can affect performance, particularly in situations of linear computation that do not have much parallelism. Perhaps most importantly, the static architecture also severely limits the execution of loops. In certain cases, the single-token-per-arc limitation means that a second loop iteration cannot begin executing until the previous one has almost completed, thereby limiting parallelism to simple pipelining and preventing truly parallel execution of loop iterations. Despite these limitations, a number of static dataflow computers have been built and studied [Davis 1978; Dennis and Misunas 1975; Dennis 1980].

2.2.2. The Dynamic, or Tagged-Token Architecture. An alternative approach was proposed by Watson and Gurd [1979], Arvind and Culler [1983], and Arvind and Nikhil [1990]. Known as the *dynamic model*, it exposes additional parallelism by allowing multiple invocations of a subgraph that is often an iterative loop. While this is the conceptual view of the tagged-token model, in reality only one copy of the graph is kept in memory and tags are used to distinguish between tokens that belong to each invocation. A tag contains a unique subgraph invocation ID, as well as an iteration ID if the subgraph is a loop. These pieces of information, taken together, are commonly known as the *color* of the token.

Instead of the single-token-per-arc rule of the static model, the dynamic model represents each arc as a large bag that can contain any number of tokens, each with a different tag [Silc et al. 1998]. In this scenario, a given node is said to be fireable whenever the same tag is found in a data token on each input arc. It is important to note that, because the data tokens are not ordered in the tagged-token model, processing of tokens does not necessarily proceed in the same order as they entered the system. However, the tags ensure that the tokens do not conflict, so this does not cause a problem.

The tags themselves are generated by the system [Arvind and Culler 1986]. Tokens being processed in a given invocation of a subgraph are given the unique invocation ID of that subgraph. Their iteration ID is set to zero. When the token reaches the end of the loop and is being fed back into the top of the loop, a special control operator increments the iteration ID. Whenever a token finally leaves the loop, another control operator sets its iteration ID back to zero.

A hardware architecture based on the dynamic model is necessarily more complex than the static architecture outlined in Section 2.2.1. Additional units are required to form tokens and match tags. More memory is also required to store the extra tokens that will build up on the arcs. Arvind and Culler [1986] provided a good summary of the architecture.

The key advantage of the tagged-token model is that it can take full advantage of pipelining effects and can even execute separate loop iterations simultaneously. It can also execute out-of-order, bypassing any tokens that require complex execution and that delay the rest of the computation. It has been shown that this model offers the maximum possible parallelism in any dataflow interpreter [Arvind and Gostelow 1977].

Another noteworthy benefit of the tagged-token model is that less care needs to be taken to ensure that tokens remain in order. For example, the pure dataflow model requires Merge operators (see Section 2.1.1) to ensure that data tokens are merged in a determinate way. In the dynamic model, however, this is not required as the tags ensure the determinacy, and so token streams can be merged arbitrarily.

The main disadvantage of the tagged-token model is the extra overhead required to match tags on tokens, instead of simply their presence or absence. More memory is also required and, due to the quantity of data being stored, an associative memory is not practical. Thus, memory access is not as fast as it could be [Silc et al. 1998]. Nevertheless, the tagged-token model does seem to offer advantages over the static model. A number of computers using this model have been built and studied [Arvind and Culler 1983; Barahona and Gurd 1985].

As stated above, the choice of target architecture can have implications on the programming of software. Depending on the model chosen, certain types of nodes, such as merge or switch nodes, are not required. Additionally, the performance of the program will be affected and some properties of the system (such as its tendency to deadlock, which can be verified under the pure dataflow model) may change subtly under certain implementations [Arvind and Culler 1986; Naggar et al. 1999]. For example, some networks that deadlock under the static model may not deadlock under the dynamic model. This is due to the pure model's theoretically valid but impractical assumptions that there are an infinite number of processing elements and infinite space on each arc [Kahn 1974].

2.3. Synchronous Dataflow

A later development in dataflow, but one that became quite widely used, was synchronous dataflow (SDF) [Lee and Messerschmitt 1987]. This is a subset of the pure dataflow model in which the number of tokens consumed and produced on each arc of a node is known at compile-time [Bhattacharyya 1996]. Under SDF, the number of tokens initially on each arc is also specified at compile-time. In this scenario, there are certain limitations that mean that some kinds of program cannot be represented. For example, loops can only be specified when the number of iterations is known at compile-time.

The advantage of the SDF approach, however, is that it can be statically scheduled [Buck and Lee 1995]. This means that it can be converted into a sequential program and does not require dynamic scheduling. It has found particular applications in digital signal processing where time is an important element of the computation [Lee and Messerschmitt 1987; Plaice 1991]. Even dataflow graphs which are not SDF in themselves may have subgraphs that are, and this may allow partial static scheduling [Buck and Lee 1995], with the rest scheduled according to the usual dataflow scheduling techniques. This has applications to coarse-grained dataflow discussed in Section 4.3.

3. EARLY DATAFLOW PROGRAMMING LANGUAGES

3.1. The Development of Dataflow Languages

With the development of dataflow hardware came the equally challenging problem of how to program these machines. Because they were scheduled by data dependencies, it was clear that the programming language must expose these dependencies. However, the data dependencies in each class of language can be exploited to different degrees, and the amount of parallelism that can be implicitly or explicitly specified also differs. Therefore, the search began for a suitable paradigm to program dataflow computers and a suitable compiler to generate the graphs [Arvind et al. 1988]. Various paradigms were tried, including imperative, logical, and functional methods. Eventually, the majority consensus settled on a specific type of functional language that became known as *dataflow languages*.

An important clarification must be made at this stage. In early publications, dataflow graphs are often used to illustrate programs. In many cases, these graphs are simply representations of the compiled code [Dennis and Misunas 1975] that would be executed on the machine, where the graph was generated either by hand or by a compiler from a third-generation programming language. Until the advent of Dataflow Visual Programming Languages in the 1980s and 1990s, it was rarely the intention of researchers that developers should generate these graphs directly. Therefore these early graphs are not to be thought of as "dataflow programming languages."

3.1.1. What Constitutes a Dataflow Programming Language?. While dataflow programs can be expressed graphically, most of the languages designed to operate on dataflow machines were not graphical. There are two reasons for this. First, at the low level of detail that early dataflow machines required, it became tedious to graphically specify constructs such as loops and data structures which could be expressed more simply in textual languages [Whiting and Pascoe 1994]. Second, and perhaps more importantly, the hardware for displaying graphics was not available until relatively recently, stifling any attempts to develop graphical dataflow systems. Therefore, traditional dataflow languages are primarily text-based.

One of the problems in defining exactly what constitutes a dataflow language is that there is an overlap with other classes of language. For example, the use of dataflow programming languages is not limited to dataflow machines. In the same way, some languages, not designed specifically for dataflow, have subsequently been found to be quite effective for this use (e.g., Ashcroft and Wadge [1977]; Wadge and Ashcroft [1985]). Therefore, the boundary for what constitutes a dataflow language is somewhat blurred. Nevertheless, there are some core features that would appear to be essential to any dataflow language. The best list of features that constitute a dataflow language was put forward by Ackerman [1982] and reiterated by Whiting and Pascoe [1994] and Wail and Abramson [1995]. This list includes the following:

(1) freedom from side effects,
(2) locality of effect,
(3) data dependencies equivalent to scheduling,
(4) single assignment of variables,
(5) an unusual notation for iterations due to features 1 and 4,
(6) lack of history sensitivity in procedures.

Because scheduling is determined from data dependencies, it is important that the value of variables do not change between their definition and their use. The only way to guarantee this is to disallow the reassignment of variables once their value has been assigned. Therefore, variables in dataflow languages almost universally obey the single-assignment rule. This means that they can be regarded as values, rather than variables, which gives them a strong flavor of functional programming. The implication of the single-assignment rule is that the compiler can represent each value as one or more arcs in the resultant dataflow graph, going from the instruction that assigns the value to each instruction that uses that value.

An important consequence of the single-assignment rule is that the order of statements in a dataflow language is not important. Provided there are no circular references, the definitions of each value, or variable, can be placed in any order in the program. The order of statements becomes important only when a loop is being defined. In dataflow languages, loops are usually provided with an imperative syntax, but the single-assignment rule is preserved by using a keyword such as *next* to define the value of the variable on the next iteration [Ashcroft and Wadge 1977]. A few dataflow languages offer recursion instead of loops [Weng 1975].

Freedom from side effects is also essential if data dependencies are to determine scheduling. Most languages that avoid side effects do so by disallowing global variables and introducing scope rules. However, in order to ensure the validity of data dependencies, a dataflow program does not even permit a function to modify its own parameters. All of this can be avoided by the single-assignment rule. However, problems arise with this strategy when data structures are being dealt with. For example, how can an array be manipulated if only one assignment can ever be made to it? Theoretically, this problem is dealt with by conceptually viewing each modification of an array as the creation of a new copy of the array, with the given element modified. This issue is dealt with in more detail in Section 6.3.

It is clear from the above discussion that dataflow languages are almost invariably functional. They have applicative semantics, are free from side effects, are determinate in most cases, and lack history sensitivity. This does not mean that dataflow and functional languages are equivalent. It is possible to write certain convoluted programs in the functional language *Lucid* [Ashcroft and Wadge 1977], which cannot be implemented as a dataflow graph [Ashcroft and Wadge 1980]. At the same time, much of the syntax of dataflow languages, such as loops, has been borrowed from imperative languages. Thus it seems that dataflow languages are essentially functional languages with an imperative syntax [Wail and Abramson 1995].

3.1.2. Dataflow Languages. A number of textual dataflow languages, or functional languages that can be used with dataflow, have been implemented. A representative sample is discussed below. (Whiting and Pascoe [1994] presented a fuller review of these languages.) Dataflow Visual Programming Languages are discussed in detail in Section 5.

—*TDFL*. The Textual Data-Flow Language was developed by Weng [1975] as one of the first purpose-built dataflow languages. It was designed to be compiled into a dataflow graph with data streams in a relatively straightforward way and supported compile-time deadlock detection. A program expressed in TDFL consisted of a series of modules, analogous to procedures in other languages. Each module was made up of a series of statements that were either assignments (obeying the single-assignment rule), conditional statements, or a call to another module. Iteration was not provided directly, as Weng could find no way to make it compatible with the single-assignment rule, but modules could call themselves recursively.

—*LAU*. Developed in 1976 for the LAU static dataflow architecture, the LAU language was developed by the Computer Structures Group of ONERA-CERT in France [Comte et al. 1978; Gelly 1976]. It was a single-assignment language and included conditional branching and loops that were compatible with this rule through the use of the *old* keyword. It was one of the few dataflow languages that provided explicit parallelism through the *expand* keyword that specified parallel assignment. LAU had some features that were similar to object-oriented languages, such as the ability to encapsulate data and operations [Comte et al. 1978].

—*Lucid*. Originally developed independently of the dataflow field by Ashcroft and Wadge [1977], Lucid was a functional language designed to enable formal proofs. Recursion was regarded as too restrictive for loop constructs, but it was realized that iteration introduced two nonmathematical features into programming: transfer and assignment. Thus, Lucid was designed to permit iteration in a way that was mathematically respectable, through single assignment and the use of the keyword *next* to define the value of the variable in the next iteration. It quickly became apparent, however, that Lucid's functional and single-assignment semantics were similar to those required for dataflow machines, and Ashcroft and Wadge [1980] brooded on the topic in literature before publishing a book in 1985 [Wadge and Ashcroft 1985] that firmly established Lucid's claim to be a dataflow language.

—*Id*. Originally developed by Arvind et al. [1978] for writing operating systems, Id was intended to be a language without either sequential control or memory cells, two aspects of the von Neumann model that Arvind et al. felt must be rejected. The resultant language had single-assignment semantics and was block-structured and expression-based. Id underwent much evolution, and later versions tackled the problem that data structures were not comfortably compatible with the single-assignment rule through the inclusion of I-structures [Arvind et al. 1989] (which are themselves functional data structures and are explained in Section 6.3).

—*LAPSE*. Developed by Glauert [1978], LAPSE was derived from Pascal and was designed for use on the Manchester dataflow machine. The language had single-assignment semantics and provided functions, conditional evaluation, and user-defined data types. It provided iteration without using any qualifying keywords to differentiate between the current and next value of the loop variable. Rather, the compiler assumed that the old value was intended if it appeared in an expression, and the next value was assumed if it appeared on the left of an assignment. Like LAU, LAPSE provided a single explicit parallel construct, *for all* for parallel array assignment.

—*VAL*. VAL was developed by Dennis starting in 1979 [Ackerman and Dennis 1979; Dennis 1977], and obeyed the single-assignment rule. A program in VAL consisted of a series of functions, each of which could return multiple values. Loops were provided by the Lucid technique [Ashcroft and Wadge 1977], and a parallel assignment construct, *for all*, was also provided. However, recursion was not provided as it was not thought necessary for the target domain. Other disadvantages

[Whiting and Pascoe 1994] included the lack of general I/O and the fact that nondeterministic programs could not be expressed.

—*Cajole*. First developed under this name in 1981 [Hankin and Glaser 1981], Cajole was a functional language designed to be compiled into acyclic dataflow graphs. It did not provide loops, but did permit recursion. Cajole was later used in a project that explored structured programming with dataflow [de Jong and Hankin 1982].

—*DL*1. Developed by Richardson [1981] to support research into hybrid dataflow architectures, DL1 was a functional language designed to be compiled into low-level dataflow graphs. This target was made more explicit than in other languages, as evidenced by keywords such as *subgraph*. The language provided for recursion and conditional execution.

—*SISAL*. Like Lucid, SISAL was not originally written specifically for dataflow machines, but found that application later. Originally developed in 1983 [Gurd and Bohm 1987; McGraw et al. 1983], SISAL is a structured functional language, providing conditional evaluation and iteration consistent with the single-assignment rule. Although it provides data structures, these are treated as values and thus cannot be rewritten like I-structures [Arvind et al. 1989]. The only parallel construct provided is a parallel loop.

—*Valid*. Designed by Amamiya et al. [1984], Valid was an entirely functional language designed to demonstrate the "superiority" of dataflow machines. It provided recursion as a key language element, but also provided functional loops using the Lucid method [Ashcroft and Wadge 1977]. A simple parallel loop construct was also provided.

The above list represents much of the population of dataflow programming languages in existence. Many are similar and the majority have (1) functional semantics, (2) single assignment of variables, and (3) limited constructs to support concurrency.

3.1.3. Using Imperative Languages with Dataflow. While the majority consensus settled on the previously mentioned dataflow languages, this does not mean that other directions were not pursued. Dataflow compilers have been built for several imperative languages [Wail and Abramson 1995]. These include Fortran, Pascal, and several dialects of C [Whiting and Pascoe 1994]. All of these approaches had to deal with the major problem of how to generate code based on data dependencies from languages that allow a lot of flexibility in this regard. Wail and Abramson [1995] confirmed that when programming dataflow machines with imperative languages, the generation of good parallel code can be extremely difficult. They also confirmed that the implementation of nonfunctional facilities, such as global variables, will reduce possible concurrency. Their main motivation for pursuing this line of research was that much software is already written in imperative languages and most programmers are already familiar with the paradigm.

In their 1982 paper, Gajski et al. [1982] offered the opinion that using dataflow languages offered few advantages over imperative languages, on the grounds that the compiler technology was as complex for one as for the other. They argued that the use of sophisticated compiler techniques and explicit concurrency constructs in an imperative language could provide the same level of parallel performance on dataflow machines as a dataflow language. While not denying the advantages of the syntactical purity of dataflow languages, they argued that these advantages do not justify the effort required for the introduction of a totally new class of programming languages.

The possibility of placing explicit concurrency constructs into dataflow languages has been largely resisted by researchers (parallel assignment/loops have been provided in some cases—see Dennis

[1977], Gelly [1976], and Glauert [1978]—but almost no other concurrency features have been included). It is probable that the explanation for this resistance is that implicit parallelism is an extremely appealing idea, and to introduce explicit concurrency constructs into their dataflow languages would destroy one of the most appealing parts of the dataflow concept.

The argument of Gajski et al. [1982] against creating specifically "dataflow" programming languages would be valid if the only justification for the pursuit of dataflow were the pursuit of improved performance through exploiting parallelism. However, as they themselves commented, dataflow languages have features that are very advantageous to the programmer. The future development of dataflow visual programming languages provides much evidence for this, where the emphasis has moved toward benefits in software engineering. Therefore, contrary to the assertions of Gajski et al. [1982] it is proposed that further research into dataflow programming languages is justified.

3.2. The Dataflow Experience in the 1980s

In the 1980s, proponents confidently predicted that both dataflow hardware and dataflow languages would supersede von Neumann-based processors and languages [Arvind et al. 1977; Treleaven et al. 1982; Treleaven and Lima 1984]. However, looking back over the 1990s, it is clear that this did not happen [Silc et al. 1998; Veen 1986; Whiting and Pascoe 1994]. In fact, research into dataflow languages slowed after the mid-1980s. In their review paper of 1994, Whiting and Pascoe [1994] reported that several dataflow researchers had concluded that dataflow had been mostly a failure—cost-effective dataflow hardware had failed to materialize. Given that the dataflow concepts looked promising, the languages were appealing, and much research effort was put into the subject, there must be good reasons for the decline of these early dataflow concepts.

It is widely believed that a main reason was that the early dataflow hardware architectures operated at a level that was too fine-grained. While von Neumann architectures operate at process-level granularity (i.e., instructions are grouped into threads or processes and then executed sequentially), dataflow operates at instruction-level granularity. This point had been recognized by 1986, when Veen [1986, p. 393] remarked that "there are signs that a deviation is also necessary from the fine-grain approach" because it led to "excessive consumption of resources." This is because it required a high level of overhead to prepare each instruction for execution, execute it, propagate the resultant tokens, and test for further enabled firings. Indeed, algorithms which exhibit a low degree of natural parallelism can execute unacceptably slowly on dataflow machines because of this degree of overhead. Veen [1986] defended these claims, arguing that the overhead can be reduced to an acceptable level by compiling techniques, but later experiences seem to demonstrate that the criticism was valid. For example, Bic [1990, p. 42] commented that "inefficiencies [are] inherent to purely dataflow systems" while Silc et al. [1998, p. 9] commented that "pure dataflow computers . . . usually perform quite poorly with sequential code."

The reason for the decline in dataflow research in the late 1980s and early 1990s was almost entirely due to problems with the hardware aspects of the field. There was little criticism of dataflow languages—other than those leveled at functional languages in general [Gajski et al. 1982]—which are still unrivaled in the degree of implicit parallelism that they achieve [Whiting and Pascoe 1994]. The dataflow execution model can be used with or without dataflow hardware, and, therefore, any decline in the hardware aspect does not necessarily affect dataflow languages, provided they have advantages on their own merit. This article contends that they do.

4. EVOLUTION OF DATAFLOW

It is not surprising, on the basis of Section 3.2, that when research into dataflow again intensified in the early 1990s [Lee and Hurson 1993], the issue of granularity was one of the key points to be addressed. One of the primary realizations that made this shift possible was the recognition that, contrary to what was popularly believed in the early 1980s, dataflow and von Neumann techniques were not mutually exclusive and irreconcilable concepts, but simply the two extremes of a continuum of possible computer architectures [Papadopoulus and Traub 1991; Silc et al. 1998].

Fine-grained dataflow could now be seen as a multithreaded architecture in which each machine-level instruction was executed in a thread on its own. At the same time, von Neumann architectures could now be regarded as a multithreaded architecture in which there was only one thread—the program itself. For example, in their survey paper, Lee and Hurson [1993, p. 286] observed that "the foremost change is a shift from the exploitation of fine- to medium- and large-grain parallelism." The primary issue in dataflow thus immediately became the question of granularity.

The result of this shift in viewpoint was the exploration of what has become known as *hybrid dataflow*.

4.1. The Development of Hybrid Dataflow

Although hybrid dataflow concepts had been explored for many years [Silc et al. 1998], it was only in the 1990s that they became the dominant area of research in the dataflow community. In their 1995 paper, Sterling et al. [1995] explored the performance of different levels of granularity in dataflow machines. Although the range of possible test scenarios is huge, they did produce a generalized graph that summarized their findings. A simplified version of this is shown in Figure 4.

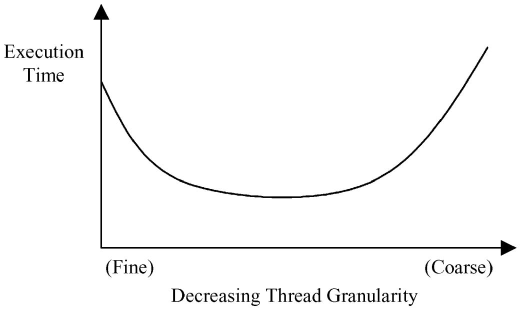


Fig. 4. Dataflow granularity optimization curve, (based on Sterling et al. [1995]).

Figure 4 indicates that neither fine-grained (as in traditional dataflow) nor coarse-grained (as in sequential execution) dataflow offers the best parallel performance, but rather a medium-grained approach should be used. This suggests that some form of hybrid—dataflow with von Neumann extensions, or vice versa—would offer the best performance. The question then was, what level of medium granularity was best?

In terms of hardware architectures, there is no universal consensus on how best to achieve this hybrid. Some approaches are essentially von Neumann architectures with a few dataflow additions. Others are essentially dataflow architectures with some von Neumann additions. (For examples see Iannucci [1988]; Nikhil and Arvind [1989]; Papadopoulos and Traub [1991]). It is not our intention to explore the hardware aspects of hybrid dataflow in depth here, as the concentration of this article is on dataflow programming, but a good summary was published by Silc et al. [1998].

While this new research was aimed at improving hardware architectures, the rejection of fine-grained dataflow and the move toward more coarse-grained execution has also freed the dataflow programming of its restriction to fine-grained execution. This allows a much wider range of research to be conducted into dataflow programming, taking advantage of these new degrees of granularity.

Essentially the now-accepted requirement for more coarse-grained execution has caused a divergence in the dataflow programming community. The first group advocates generating fine-grained dataflow graphs as before, but they then propose analyzing these graphs and identifying subgraphs that exhibit low levels of parallelism that should always execute in sequence. These nodes are grouped together into segments. Thus, when the first node in the segment is fired, the remaining nodes can be fired immediately. They are still executed in a fine-grained manner, but the costly token-matching process is avoided for the subsequent nodes in the sequence, saving time and resources. This approach is termed *threaded* dataflow [Silc et al. 1998].

The second group advocates dispensing with fine-grained dataflow execution, and instead compiling the subgraphs into sequential processes. These then become coarse-grained nodes, or macroactors. The graphs are executed using the traditional dataflow rules, with the only difference being that each node contains, for example, an entire function expressed in a sequential language as opposed to a single machine-level instruction. This second approach is usually termed *large-grain* dataflow [Silc et al. 1998].

It was noted as early as 1974 that the mathematical properties of dataflow networks are valid, regardless of the degree of granularity of the nodes [Arvind et al. 1988; Jagannathan 1995; Kahn 1974; Lee 1997; Sterling et al. 1995], and, therefore, the hybrid approaches to dataflow programming do not in any way compromise the execution model. Both the threaded and large-grain approaches are exciting developments but it is the latter that offers the most potential for improvements to dataflow programming.

4.2. Threaded Dataflow

The threaded dataflow approach takes advantage of the fact that dataflow program graphs display some level of sequential execution. For example, in the case where the output of one node goes into the next node, the two nodes could never execute in parallel when they are operating on a single wave of data. Therefore, there is little point in scheduling them to two different processors. Under fine-grained dataflow, the output token from the first node must be mapped back through the system, added to the input arc for the second node, which must then wait to be fired. It is much more efficient to place the two instructions in a single execution quanta, so that the output of the first node can be immediately used by the second [Bic 1990].

This principle was used by Bic [1990], who proposed analyzing a dataflow graph and producing sequential code segments (SCS), which are nodes that are in a chain and cannot be executed in parallel. Under the modified execution model, the granularity is at the SCS level. However, other than the fact that the execution of the first instruction in an SCS causes the rest of the chain to be executed, the model obeys the standard dataflow rules. Bic [1990] also proposed a method to automatically identify the SCSs without programmer intervention.

The advantage of the threaded dataflow approach is that those parts of the dataflow graph that do not exhibit good potential parallelism can be executed without the associated overhead, while those that do show potential parallelism can take advantage of it. In their study of this approach, Papadopoulus and Traub [1991] confirmed these conclusions, although they warned that it is not wise to carry the line of sequentiality too far. An analysis of these proposals undertaken by Bohm et al. [1993] demonstrated that medium-grained dataflow did indeed improve performance, as predicted by Sterling et al. [1995].

A key open question is how best to partition programs into threads and what degree of granularity is best [Bohm et al. 1993; Lee and Hurson 1994]. The Pebbles group in the U.S. is examining the relationship between granularity of parallelism and efficiency in hybrid dataflow [Bohm et al. 1993; Najjar et al. 1994], although the group is primarily concerned with large-grain dataflow. Another consequent benefit of this area of research has been that it is easier to encode certain functions (e.g., resource management) if some sequential execution is permitted [Lee and Hurson 1994].

4.3. Large-Grain Dataflow

Large-grain dataflow can begin with a fine-grained dataflow graph. This dataflow graph is analyzed and divided into subgraphs, much like the threaded approach. However, instead of remaining as groupings of associated nodes, the subgraphs are compiled into sequential von Neumann processes. These are then run in a multithreaded environment, scheduled according to the usual dataflow principles. The processes are termed macroactors [Lee and Hurson 1994]. Much recent work has been done in the area of large-grain dataflow systems and it offers a great opportunity for improvements to the field of dataflow programming.

One important point is that, since the macroactors in large-grain dataflow are sections of sequential code, there is no reason why these have to be derived from fine-grained dataflow graphs. The macroactors could just as easily be programmed in an imperative language, such as C or Java. Each macroactor could represent an entire function, or part of a function, and could be designed to be used as off-the-shelf components. It is the fact that the macroactors are still executed according to dataflow rules that lets this approach retain the clear advantages of dataflow, but solve the high-overhead problem of fine-grained dataflow. Research has been conducted into the best degree of granularity by Sterling et al. [1995], who found a medium-grained approach to be optimal.

A related field is the "flow-based programming" methodology advocated by Morrison [1994]. Morrison advocated the use of large-grain components, expressed in an imperative language, but linked together in a way reminiscent of dastaflow. Although it is not dataflow—it does not strictly obey the dataflow firing rules—Morrison's proposals do suggest features that could be incorporated into dataflow, including greatly simplified schemes for providing iteration and the use of substreams within streams. The book emphasizes the benefits of these principles to business software, and to software engineering in general.

The key benefit of Morrison's [1994] approach is a much reduced development time. Empirical evidence of this is offered in his book, where real-life experience of a large piece of software developed with flow-based techniques is cited. The approach led to a considerable savings of time and effort on the part of the programmers, particularly when it came to modifying the program after initial completion. Morrison expounded at length on the benefits of the stream-based, large-grained modular approach to business software engineering in particular. If these concepts were applied to large-grain dataflow, the advantages of both dataflow execution and Morrison's graphical component-based software could be merged.

Similar techniques have already been used in one key area—digital signal processing [Bhattacharyya 1996; Naggar et al. 1999]. Many signal-processing environments, such as Ptolemy [Bhattacharyya 1996], operate by letting the user connect together components, each of which performs a medium-grained programming task. The whole network is essentially a dataflow network. Some work has been done in formalizing such networks [Lee and Parks 1995]. It has been noted that these principles were used by the signal-processing community before being formalized in research, and have, therefore, already been demonstrated to be beneficial to software engineering [Lee and Parks 1995].

5. RECENT DEVELOPMENTS IN DATAFLOW PROGRAMMING LANGUAGES

5.1. Introduction

The last major reviews of dataflow programming languages were Hils [1992] and Whiting and Pascoe [1994]. In the decade since then, the field of dataflow has expanded and diverged to include many disparate areas of research. However, the focus of this section is strictly on programming languages that are based upon the dataflow execution model. Indeed, there are some languages that have the appearance of dataflow, but upon examination, it is clear that they are sufficiently different from the pure dataflow execution model to make such a label questionable, for example, JavaBeans.

Since the majority of developments in dataflow programming languages in the past decade have been in the field of visual programming languages, this section also concentrates on visual programming languages. It should be stressed that the textual dataflow languages detailed in previous sections still exist and are being developed, although most of the current research in that area is in the field of hardware and compilation technology. Since the emphasis in this article is on software engineering rather than hardware, these issues are beyond the scope of this section. Hardware issues are mentioned only insofar as they have affected the development of dataflow languages.

As has already been outlined in the previous section, the major development in the past 10 years in dataflow has been the move away from fine-grained parallelism toward a more coarse-grained approach. These approaches ranged in concept from adding limited von Neumann hardware to dataflow architectures, to running dataflow programs in a multithreaded manner on machines that were largely von Neumann in nature.

To some extent, dataflow languages evolved to meet these new challenges. A greater emphasis was placed upon compiling dataflow programs into a set of sequential threads that were themselves executed using the dataflow firing rules. However, these changes did not have a major effect upon the languages themselves whose underlying semantics did not have to change.

From a software engineering perspective, however, the major development in dataflow in the past 15 years has been the growth of dataflow visual programming languages (DFVPLs). Although the theory behind DFVPLs has been in existence for many years, it is only the availability of cheap graphical hardware in the 1990s that has made it a practical and fruitful area of research.

Investigations of DFVPLs have indicated many solutions to existing problems in software engineering, a point which will be expanded upon below. It has also led to the introduction of new problems and challenges, particularly those associated with visual programming languages in general [Whitley 1997], as well as continuing problems, such as the representation of data structures and control-flow structures [Auguston and Delgado 1997; Ghittori et al. 1998; Mosconi and Porta 2000]. Research has been fairly intense in the past decade, and it is the subject of this section to identify some of the main trends in dataflow programming over this period.

5.2. The Development of Dataflow Visual Programming Languages

In Section 3, textual dataflow languages were discussed, and much of the research into dataflow hardware utilizes these textual languages. The "machine" language of programs designed to be run on dataflow hardware architectures is the dataflow graph. Most textual dataflow languages were translated into these graphs in order to be scheduled on the dataflow machine.

However, early on it was realized that these graphs could have advantages in themselves for the programmer [Davis 1974; Davis and Keller 1982]. Graphs allow easy communication of ideas to novices, allowing much more productive meetings between the developer and the customer [Baroth and Hartsough 1995; Morrison 1994; Shürr 1997]. In addition, a range of research into VPLs has indicated the existence of significant advantages in a visual syntax [Green and Petre 1996], for example, dynamic syntax and visualization [Hils 1992; Shizuki et al. 2000]. The fact that several dataflow environments have been the basis of successful commercial products adds weight to this case [Baroth and Hartsough 1995]. Finally, research has shown that most developers naturally think in terms of dataflow in the design phase, and DFVPLs remove the paradigm shift that is forced on a programmer when entering the coding phase. Indeed, DFVPLs arguably remove this distinction altogether [Baroth and Hartsough 1995; Iwata and Terada 1995].

Researchers published papers on DFVPLs intermittently in the 1980s. Their ideas were intriguing and showed great promise, but were restricted by the expense and low diffusion of graphical hardware and pointing devices. Davis and Keller [1982] recognized the now universally accepted trend toward more graphically based computer systems, and made the argument that textual languages could be completely replaced by graphical ones in the future. Although this prediction has not fully come to pass, they judiciously proposed that human engineering rather than concurrent execution would become the principal motivation for developing dataflow visual programming languages, a motivation that has indeed been at the fore of more recent DFVPL research.

5.2.1. Early Dataflow Visual Programming Languages. In the 1970s, Davis [1974, 1979] devised *Data-Driven Nets* (DDNs), a graphical programming concept that was arguably the first dataflow visual language (as opposed to a graph used purely for representation). In DDN, programs are represented as a cyclic dataflow graph with typed data items flowing along the arcs which are FIFO queues. The program is stored in a file as a parenthesized character string, but displayed as a graph. The language operates at a very low level and, in fact, Davis [1978] commented that it was not the intention that anyone should program directly in DDNs. Nevertheless, they illustrated key concepts such as the feasibility of providing iteration, procedure calls, and conditional execution without the use of a textual language.

By the early 1980s, Davis had developed a more practical, higher-level DFVPL known as *GPL* (*Graphical Programming Language*). Davis and Lowder [1981] contended that text-based programming languages lacked intuitive clarity and proposed going further than using graphs as a design aid by creating an environment in which the program is a graph. GPL was also an attempt to create a higher-level version of DDNs [Davis 1979; Whiting and Pascoe 1994]. In the GPL environment, every node in the graph was either an atomic node or could be expanded to reveal a sub-graph, thereby providing structured programming with top-down development. These subgraphs could be defined recursively. Arcs, in the graph were typed and the whole environment had facilities for debugging, visualization and text-based programming if desired. The lack of suitable graphical hardware for their system was the main reason for a lack of rapid development of these concepts.

In the early 1980s, researchers Keller and Yen [1981] developed FGL, independently from Davis. FGL stands for *Function Graph Language*, and was born from the same concept of developing dataflow graphs directly. Unlike the token-based dataflow model of GPL, FGL was based around the structure model, of which Keller was a proponent [Davis and Keller 1982] (see Section 2.1.2). Under this model, data is grouped into a single structure on each arc rather than flowing around the system. In other regards, FGL was similar to GPL in its support for top-down stepwise refinement. The relative advantages and disadvantages of GPL and FGL mirror those of the token-flow model and structure model, respectively.

Shortly afterwards, the Grunch system was developed by de Jong et al. [1982], the same researchers who created the Cajole textual dataflow language [Hankin and Glaser 1981]. While not a programming language in the proper sense, it was a graphical overlay for Cajole that allowed the developer to graphically express a dataflow program using stepwise refinement, and then use the tool to convert the graph into Cajole. The actual conversion was performed by an underlying tool called *Crunch*. The development of Grunch supported the claims of Davis and Keller [1982] that software engineering could be as much a motivation for pursuing graphical dataflow as the pursuit of efficient parallelism.

5.2.2. More Recent Dataflow Visual Programming Languages. Interestingly, from the mid-1980s on, further development of DFVPLs often came from different sources than direct research into dataflow. Indeed, industry played a part in this phase of development. The most common source was signal- and image-processing, which lends itself particularly well to a dataflow approach [Buck and Lee 1995]. Therefore, many DFVPLs were produced to solve specific problems and utilized dataflow because it provided the best solution to the problem. As Hils [1992] commented, DFVPLs in this period were most successful in narrow application domains and in domains where data manipulation is the foremost task.

Hils [1992] provided details of 15 languages developed in the 1980s and very early 1990s that could be classed as DFVPLs. In order to avoid repetition, only two examples of these are discussed here. NL, a significant language that appeared after Hils wrote his paper, is also described.

LabView is a well-known DFVPL developed in the mid-1980s to allow the construction of "virtual" instruments for data analysis in laboratories. As such, it was intended for use by people who were not themselves professional programmers. A program in LabView is constructed by connecting together predefined functions, displayed as boxes with icons, using arcs for data paths. Each program also has a visual interface to allow the design of the virtual instrument. Components that have a visual representation appear both in the interface and the program, whereas functions only appear in the program window. The whole program is executed according to the dataflow firing rules. LabView makes the programming experience less cumbersome by providing iterative constructs and a form of stepwise refinement whereby programmers can produce their own function nodes.

Empirical evidence reported by the Jet Propulsion Laboratory [Baroth and Hartsough 1995] has shown a very favorable experience with LabView when used for a large project, compared to developing the same system in C. In particular, they found that the DFVPL led to a significantly faster development time than C, mainly due to the increased communication facilitated by the visual syntax. An example of a program written in LabView is shown in Figure 5. As well as its demonstrated and continuing industrial successes, LabView has proved particularly popular with researchers [Ghittori et al. 1998; Green and Petre 1996].

ProGraph was a more general-purpose DFVPL than LabView, and involved combining the principles of dataflow with object-oriented programming. The methods of each object are defined using dataflow diagrams. Like LabView, ProGraph includes iterative constructs and permits procedural abstraction by condensing a graph into a single node. ProGraph has also been used as a subject in research [Cox and Smedley 1996; Green and Petre 1996; Mosconi and Porta 2000]. Example screenshots of ProGraph programs can be found in Mosconi and Porta [2000].

In the mid 1990s, the language NL was developed by Harvey and Morris [1993, 1996], along with a supporting programming environment. It is fully based on the dataflow model of execution. NL has an extended typing system, whereby arrays can behave as arbitrarily long lists, to the point of being infinite. It provides an ingenious method of control flow, through combined use of "block" and "guard" nodes. For example, a guard node may contain a condition that, if evaluated to true, causes its associated block node to be executed. Sequences of guard nodes can be created, and once one guard has been executed, all others are ignored. This has the advantage of reducing screen clutter and making

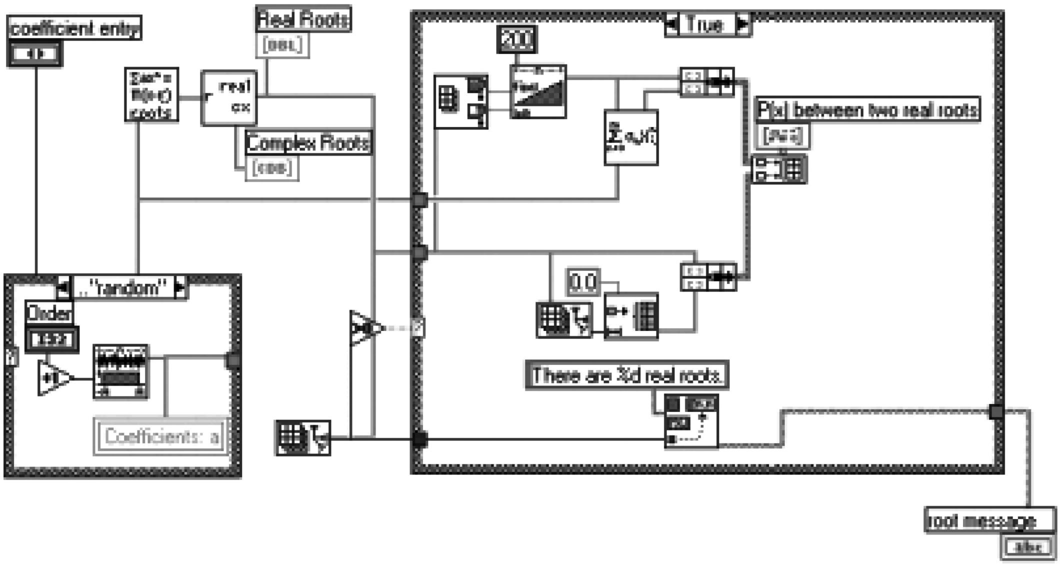


Fig. 5. Example program in LabView designed to find the real roots of a quadratic equation.

the flow of control more explicit. Loops are supported by a related method.

The NL environment reported in Harvey and Morris [1996] features a visual debugger. Screenshots of typical NL programs can also be found in this article. Programmers sees their program in the same way that they developed it and can choose to step one firing at a time, or use breakpoints. A node that is firing is highlighted, and placing the cursor over a port allows them to examine and change values. When it comes to loops, a slider allows the programmers to select any of the iterations that are taking place and examine them in any order.

5.2.3. Dataflow as a Coordination Language. Gelernter and Carriero [1992] emphasized the concept of the coordination language, that is, the concept that programming consists of two tasks, not one: computation, which specifies what is to be done, and coordination, which specifies how the computations are to proceed. They argued that the need for such a distinction was becoming more necessary with the advent of distributed and heterogeneous computer systems. The proposal was for a separation of the computation language and the coordination language.

Dataflow researchers have taken this idea on board. Indeed, since dataflow graphs explicitly express the relationships between computations, it is clear that dataflow is a natural coordination language. While few researchers have gone so far as to create an entirely independent general-purpose co-ordination language based on dataflow ideas, many have produced DFVPLs that strongly display the distinction. For example, the Vipers DFVPL [Bernini and Mosconi 1994] is a coordination language where the nodes in the graph are expressed using the language Tcl.

Morrison's [1994] flow-based programming concept, while it does not strictly obey the rules of dataflow, describes a system where nodes are built in arbitrary programming languages which the programmer arranges using a single network editing environment. Morrison [1994] reported empirical evidence that appears to support his assertion that this method is practical in real-world situations.

An excellent example of such a system is Granular Lucid (GLU) which was developed by Jagannathan [1995]. It is based upon Lucid, with the key addition that functions are defined in a foreign, probably sequential, language such as C. Data types are also of a foreign format. Since Lucid itself is a textual dataflow language, GLU allows a much more coarse-grained approach to dataflow by the programmer. Instead of primitive operations being executed in a fine-grained manner, this allows the rules of dataflow to be applied to a much coarser granularity. Jagannathan [1995] went on to show how this degree of granularity achieves performance similar to conventional parallel languages and concluded that using dataflow programming languages to develop applications for conventional parallel processors is feasible.

All of the above-mentioned languages are examples of how dataflow programming may be moved to a higher level of abstraction. For example, a language in which entire functions are enclosed within a node could be envisaged, while the nodes themselves are executed, using the dataflow semantics. This would be a development of the ideas put forward by Bernini and Mosconi [1994] and by Rasure and Williams [1991].

With the recent trend toward heterogeneous distributed systems and component-based programming, it is believed that thinking of dataflow as a coordination language has much merit and one that deserves further investigation.

5.3. Assessment of Visual Dataflow Programming Environments

The power of any visual programming language depends more heavily upon its environment than its text-based counterparts. The ease with which tasks can be performed has a large bearing on how it compares to other languages. Those who have used DFVPLs in industry have commented that the visual nature is an essential component of the language, not simply an interface, and that without the visualization tools offered by the environment, DFVPLs would have limited use [Baroth and Hartsough 1995].

In keeping with this fact, the trend in the late 1990s has been toward developing programming environments in tandem with the DFVLPs that they use. Indeed, so tightly have the two become that it has become difficult to distinguish where the language ends and the environment begins. Therefore, this section necessarily overlaps with language issues. Many of the advantages of DFVPLs are advantages of their environments as much as of the language.

Burnett et al. [1995] discussed what is needed in order to scale up a visual programming language to the point of being a practical proposition for a sizeable real world project. They came up with a list of four things that VPLS are trying to achieve. These are the reduction in key concepts, such as pointers; a more concrete programming experience, such as exploring data visually; explicit definitions of relationships between tasks; and immediate visual feedback.

DFVPLs have the potential to achieve all of these to some degree. The dataflow graph itself is an ideal example of an explicitly defined relationship, and it is true that they have a smaller set of key concepts than their textual counterparts. A number of the languages reviewed by Hils [1992] feature a high degree of liveness, that is, immediate visual feedback, with one, VIVA [Tanimoto 1990], allowing programmers to dynamically edit the programs while it is running visually in front of them. Finally, the visual debugger in Harvey's NL environment [Harvey and Morris 1996] is a good example both of the immediate feedback of information and the visual exploration of data.

Of course, it is more difficult to measure how well a DFVPL meets the criteria that it sets out to achieve. There are few metrics available in literature at this stage, although Kiper et al. [1997] offered one set of subjective metrics for measuring VPLs in general. Their criteria included its scalability, its ease of comprehension, gauging the degree of visual nature of the language, its functionality, and its support for the paradigm. The first of these points, that of scalability, has been answered to some degree by Burnett's work, mentioned above [Burnett et al. 1995].

An attempt to measure the comprehension of dataflow languages was made by Green and Petre [1996]. They studied ProGraph and LabView at length, and concluded that they had clear advantages. They drew the following interesting conclusions regarding the current state of DFVPLs:

—that DFVPLs allow the developer to proceed with design and implementation in their own order, thus making the design process freer and easier;

—that secondary notation could be utilized much more than it currently was;

—that more work needed to be conducted on incorporating control-flow constructs;

—that the effectiveness of program editors remained to be investigated in literature;

—that the problem of real estate was not as major as many assume it to be.

Further feedback on what is needed in DFVPLs was provided by Baroth and Hartsough [1995]. Having used a DFVPL for a real-world project, they concluded that the advantages offered lie more toward the design end of the software lifecycle, and less in the later stages of coding. They found increased communication between developer and customer, commenting, "We usually program together with the customer at the terminal, and they follow the data flow diagrams enough to make suggestions or corrections in the flow of the code. It is difficult to imagine a similar situation using text-based code" [Baroth and Hartsough 1995, p. 26]. The development time improvement in this case was a factor of 4.

By contrast, Baroth and Hartsough [1995] commented that the provision of software libraries, while speeding up coding, is merely a case of "who can type faster," and is not an advantage in itself. And so, the issue of the provision of a library of nodes is not a major one for DFVPLs. Already, a DFVPL can be used with a very primitive set of nodes and the provision of nodes that can be built up from these primitives is really an issue for the vendor of the programming environment, not for academia.

A point that Baroth and Hartsough [1995] were keen to stress was that visualization, and animation in particular, is absolutely essential to making the tool useful. Indeed, they went so far as to comment that "the graphics description of the system without the animation would not be much more than a CASE tool with a code generator" [Baroth and Hartsough 1995, p. 28].

A final point made by Baroth and Hartsough [1995] was that the boundaries between the requirements, design, and coding phases of the software lifecycle collapse and blend into one another. This appears to be both an advantage and a disadvantage. It is a problem in that the existing methodologies in place were unable to support the tool and this led to an inability to assess the progress in the project. On the other hand, the single phase allows the customer to be involved at all stages, reducing the prospect for expensive mistakes, and also reducing development time.

It is the previously mentioned emphasis on the design phase that prompted the development of Visual Design Patterns (VDPs) by Shizuki et al. [2000; Toyoda et al. 1997]. Under the VDP approach, the user is equipped with generic design patterns of common task layouts. Developers choose a VDP to suit their needs and then insert specific components into the holes in the pattern in order to produce an actual implementation. The concept has been introduced into the KLIEG environment and demonstrated in the literature. In their more recent paper, Shizuki et al. [2000]. extended the idea to include the possibility that the use of VDPs could help to focus a smart environment on the specific aspects of dataflow execution that a developer is likely to be interested in.

Animation is an important concept that was highlighted above. There is a significant difference between viewing a program graphically, and viewing it dynamically. The animation of executing dataflow programs is an exciting topic, but one in which research has only recently been undertaken in detail. A good example is Shizuki et al. [2000], which explored how a large program can be animated for a programmer. This addressed the problems of how to view multiple layers at once, how to view different areas of the program at once, how to change focus rapidly so as to avoid loss of concentration, and how to create sufficiently smooth animation that will not appear disjointed to the developer. The solution proposed is a smart, multifocal, fisheye algorithm. Much research deserves to be done in this area.

On the basis of this discussion, the following conclusions concerning Dataflow Visual Programming Environment can be drawn:

—In a DFVPL there is a blur in the distinction between language and environment.

—In addition, DFVPLs tend to significantly blur the distinctions between the requirements, design, coding, and testing phases of the software lifecycle.

—This blurring offers the opportunity for rapid prototyping.

—The design phase benefits the most from the use of DFVPLs over textual languages.

—The animation offered by a DFVPL environment is vitally important to its usefulness.

—The dataflow semantics of DFVPLs are intuitive for nonprogrammers to understand and thus improve communication between the customer and the developer.

—The library of functions included with a DFVPL is not a major factor in productivity.

—Key areas requiring work include the use of secondary notation, and control-flow constructs.

6. OPEN ISSUES IN DATAFLOW PROGRAMMING

Dataflow programming is an active area of research, and many problems remain open. Four of these issues are discussed in more detail in this section:

—the provision of iteration in textual dataflow languages,

—iteration structures in DFVPLs,

—the use of data structures,

—nondeterminism.

6.1. Iteration in Textual Dataflow Languages

Most dataflow programming languages provide loops of some form, but the way in which loops are expressed as a dataflow graph is quite different from most other representations of iteration. The problem arises because iteration does not fit neatly into the functional paradigm, as it involves repeated assignment to a loop variable and sequential processing. Nevertheless, most dataflow researchers recognized that programmers' demands made it necessary to provide iteration [Ackerman 1982] and worked on ways to make it mathematically respectable [Ashcroft and Wadge 1977]. Ways of making it efficient were also studied [Ning and Gao 1991]. It should be noted that many dataflow languages provide iteration through tail-recursion. However, as this is usual practice in functional languages, this section deals specifically with the more explicit iterative constructs.

The exact syntax of the various solutions offered differed, but they were all fundamentally the same. The idea was to think of the body of an iteration as being executed in an environment where the loop variable had a certain value that remained the same throughout the iteration. Thus, a single pass of the loop can be regarded as a set of definitions like any other. The loop variable is updated by using an identifier such as "NEW" to refer to the value that the loop variable will have on the next iteration. For example,

$$\text{NEW X} = \text{X} + 1;$$

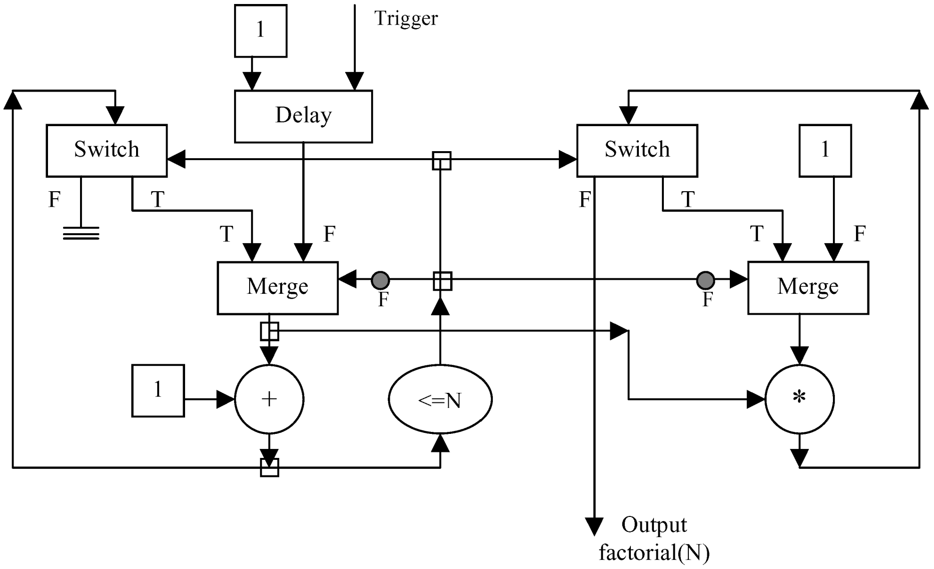


Fig. 6. Dataflow graph representing the factorial program.

As the value of X has not actually been changed by this statement, this is a mathematically acceptable way of representing iteration. When the loop has completed the iteration, the value of NEW X is assigned to X, but again, this is acceptable since all that is required is that the value of X remain unchanged during the single iteration. Some languages use an "old" keyword to achieve the same effect.

A piece of code to calculate factorial(N) by iteration, when translated into the functional loop favored by dataflow programming languages, looks like this:

```
LOOP WITH i = N, fact = 1
        NEW fact = fact * i;
        NEW i = i − 1;
WHILE i > 1;
```

In this code, the values of "fact" and "i" are defined functionally, using the loop. They are modified using the keyword "NEW." Note that the definitions of "NEW fact" and "NEW i" can be placed in any order. If the definition of "NEW i" were placed first, the definition of "NEW fact" would still be valid because the original value of i is unchanged until the end of the iteration. Note also that the value "fact" must be declared as a loop variable in the dataflow version of the loop because that is the only way that a variable can be assigned multiple times in the manner required by iteration.

While this code definitely looks different from the imperative example, it does, nevertheless, retain a strong imperative feel and could be used more intuitively by programmers when compared to tail-recursion.

Of course, this code has to be translated into a dataflow graph before it can be executed. While a loop in a dataflow graph can look complicated, most loops can be coded in the same way. Figure 6 shows a dataflow graph that could result from the above dataflow code example.

It cannot be denied that this representation is much less succinct than the text-based loop. However, the point is not that a loop can be drawn directly as a graph, but that the text-based loop can be converted into a well-behaved dataflow graph. Few dataflow researchers would expect any programmer to manually generate the graph shown in Figure 6. This also illustrates one of the failings of early graphical dataflow languages. However, as we shall note in the next section, recent development in dataflow research permit programs to be specified graphically without this level of detail.

In Figure 6, the rectangles marked "Switch" and "Merge" operate as explained in Section 2.1.1. The "Delay" gate simply waits until data appears at both inputs before outputting the data on the left arc and discarding the data on the right arc. This acts as a trigger, preventing the loop from repeatedly executing ad infinitum. With the Delay gate, a single token passed down the trigger arc will cause one execution of the loop. The squares with "1" in them are constants, that repeatedly generate tokens with the value "1." The circles are nodes that perform operations. These operations produce either numerical or Boolean results. The small grey circles labeled "F" signify tokens that are defined to be present on the given arcs when the program first activates. Three horizontal parallel lines denote a sink, which destroys any tokens that fall into it. A small open square at a crossing of two arcs indicates that the arcs are joined. In all other cases, arcs pass over each other without being joined.

If executed under the pure token-based dataflow model, with $N = 3$, the 32 separate firings necessary to complete the execution are performed in 14 time units, with parallelism in each time unit of either two or three instructions. The left-hand side of the graph produces a sequence of tokens representing the counter "i," starting at 1. The right-hand side produces a series of tokens representing the accumulating factorial by multiplying the previous factorial token by a token from the first sequence each time. The node in the center of the graph halts the feedback once the value of $i \leq N$ becomes false, and the Switch gates are used to output the completed values. The value of "i" is discarded, while the value of the factorial is sent out of this portion of the graph.

When executed under the pure token-based dataflow model, as above, the graph exhibits some pipelining within one instance of the loop, as the next iteration can begin before the previous one has completed. However, the necessary conditional check delays the execution of the Switch nodes that could otherwise begin to execute sooner. When the situation of several loops being executed at once is considered, that is, by several triggers arriving simultaneously, the pure model permits very little overlap of separate loop instances: in this case, the maximum overlap is under 10%. This is because the token that will begin the next loop is delayed by the Merge gate until a false token arrives, and this only occurs when the previous loop has completed.

When executed under the dynamic model, the loop does not provide any more pipelining of iterations within a loop, but it does provide excellent overlap of separate iterations. In fact, they can occur simultaneously and independently. Under the dynamic model, the "false" tokens that are initially placed at the two Merge gates will be present initially for each separate instance of the loop, rather than having to be generated by the previous loop as it completes. It should also be noted that other loops, such as those which populate arrays, do not have the pipelining problems mentioned above.

As illustrated above, some loops in dataflow graphs have the potential to limit concurrency. However, the use of alternative models of execution can limit this restriction. Although it is not totally natural in functional languages, iteration has been accepted as necessary by most researchers. Indeed, Whiting and Pascoe [1994, p. 53] commented that "the introduction of this form of loop construct . . . was responsible for much of the acceptance of data-flow languages." The efficient execution of dataflow loops has been a subject of active and ongoing research [Bic et al. 1995; Ning and Gao 1991; Yu and D'Hollander 2001].

6.2. Iteration in Dataflow Visual Programming Languages

Although iteration remains an open question in DFVPLs as well as textual dataflow languages, it is a different kind of problem. Here the problem is how to express a repetitive structure in a graphical model that does not naturally allow such structures. Figure 6 shows how a loop looks under the pure dataflow model. Few programmers would wish to construct such a graph, and, if they did, it would be unclear and error-prone. It has long been recognized that a practical DFVPL must provide a better way to support iteration. The question has been what constructs are most appropriate for expressing iteration. It should also be noted that iteration is merely an example of the wider issue of how to express control-flow constraints in DFVPLs. However, since iteration is arguably the most important and heavily researched problem, this section concentrates on it.

A key recent article on this topic was Mosconi and Porta [2000], and we do not intend to reproduce their review. Instead, each of five examples of iteration constructs will be described briefly.

—*Show-And-Tell*. Show-and-Tell [Kimura and McLain 1986] was an early dataflow visual language designed for children. In its approach to iteration, a special node is used to enclose an area of code that is to be executed iteratively. Each loop box has what is known as a *consistency* check. Data can only flow through a node if it is consistent. If the consistency check evaluates to false, the node becomes inconsistent, and execution of the loop stops. The loop has the same number of inputs, as outputs and data is fed back from the outputs into the inputs, as long as the box is consistent. When it becomes inconsistent, the data is ejected to the rest of the graph.

For example, a loop might contain one input that identifies the number of iterations required. This value is decremented and sent to the output during each iteration. The consistency check is that this value is greater than zero. Thus, when the iteration count reaches zero, the loop stops executing. Screenshots of Show-and-Tell loops can be found in Mosconi and Porta [2000].

—*LabView*. LabView, a commercial product, has two kinds of loop, a FOR loop and a WHILE loop [LabView 2000]. Like Show-and-Tell, a FOR loop is a special node that encloses all of the nodes to be executed iteratively. Unlike Show-and-Tell, it has an additional input port that specifies how many times the loop is to run. All other values that are output ports conceptually reenter on identical input ports. Another port visible only inside the loop specifies the current value of the loop variable.

The WHILE loop operates in a similar way, except that it does not have the loop variable. Instead, it has a port only visible inside the loop that terminates after the current iteration once it receives a value of "false." A construct unique to LabView allows the timings of the loop to be specified, for example, loop every 250 ms. This is due to its application of reading scientific instruments. A LabView program is shown in Figure 5. Further screenshots can be found in Mosconi and Porta [2000].

—*Prograph*. In Prograph [Cox et al. 1989], any user-defined node that has the same number and type of inputs as outputs can be deemed to be a loop. Its icon changes to illustrate this fact. Prograph provides a special "terminate" node for use within a loop. When the condition specified within the terminate node is satisfied, the iteration is terminated after the current iteration is complete.

—*Cantata*. Cantata [Rasure and Williams 1991] is a coarse-grained language in which nodes contain entire functions, rather than just a primitive operation. Its approach is to conceal the entire loop within one node. Each input is designated a name by the programmer, who also specifies either a loop variable and bounds, or a WHILE-condition, using the names. The programmer then sets up a series of assignments that are to take place within each loop. The node then executes the loop internally.

Note that this is far removed from pure dataflow philosophy. For example, a loop assignment may contain the expression $j = j + 1$, a statement which traditionally makes no sense in a dataflow language. Examples of

Cantata programs can be found in Mosconi and Porta [2000].

—*VEE*. In contrast with Cantata, loops in VEE [Helsel 1994] are expressed most closely to the pure dataflow model, that is, through cycles in the graph. However, the model has been augmented with a number of additional nodes in order to simplify the appearance of the cycles. In a FOR loop, a special FOR node generates a series of indexes between a range that are queued. The programmer does not need to worry about incrementing and feeding back the loop variable, being free instead to concentrate on the values being calculated. A WHILE loop can be set up by using three related nodes. The UNTIL BREAK node repeatedly activates the graph it is connected to, until the graph activates a related BREAK node which halts the repetition. Data arriving instead at the NEXT node triggers the next iteration.

Mosconi and Porta [2000] concluded their paper by proposing a syntax that is consistent with the pure dataflow model. They were keen to stress that they were not proposing their syntax for actual use, but to prove that practical iteration is possible without sacrificing the pure semantics of the model. Their loop system, implemented as part of the Vipers environment [Ghittori et al. 1998], includes cycles, but is simplified by the use of enabling signals. They also demonstrated a way to collapse an iteration into a single node without sacrificing the pure model.

All of these approaches have advantages and disadvantages. Some, such as Cantata, introduce imperative structures that are inconsistent with dataflow, although Cantata also offers the simplest loops in terms of visual syntax. Others, such as VEE, involve relatively complex graphs. All of them suffer from the inability to dynamically express the concept of a repetitive loop with a static icon. The whole area of control-flow constructs, and iteration in particular, remains an open topic in DFVPLs.

6.3. Data Structures

One of the key issues in the drive for an efficient implementation of dataflow is that of data structures. Whiting and Pascoe [1994] commented that "data structures sit uneasily within the data-flow model" (see also Treleaven et al. [1982] and Veen [1986]). However, they went on to note that much research has been undertaken in this area and that a number of quite successful solutions have been proposed, most notably I-structures [Arvind et al. 1989].

The "pure" token model of dataflow states that all data is represented by values that, once created, cannot be modified. These values flow around the dataflow graph on tokens and are absorbed by nodes. If a node wishes to modify this value, it creates a new token, containing new data which is identical to the original data, except for the element that had to be altered. Some of the earliest dataflow languages that had support for data structures worked in this way [Davis 1979]. If this way of treating data as values rather than variables were not part of the token model of dataflow, then the single-assignment rule would have been violated and thus the data-dependent scheduling of the entire graph would be compromised.

While this conceptual view of data structures is perfectly fine for the theoretical study of dataflow, and perhaps even for dataflow graphs that deal only with primitive data types, this approach is clearly unsatisfactory for graphs that require the use of data structures. With the era of structured programming, followed by the era of object-oriented programming, the idea of software development without the use of data structures is virtually incomprehensible. Thus, any practical implementation of dataflow must include an efficient way of providing data structures, although it should be stated that some languages designed for research purposes solved the problem by not providing data structures at all [Hankin and Glaser 1981].

6.3.1. Dennis's Method. Dennis [1974] was the first to provide realistic data

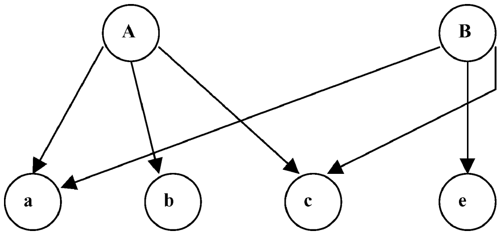


Fig. 7. Showing the effect on Dennis's data heap of modifying a value.

structures in a dataflow context by proposing that the tokens in the dataflow program hold not the data itself, but rather a pointer to the data (see also Davis and Keller [1982]). He devised a memory heap in the form of a finite, acyclic, directed graph where each node represents either an elementary value or a structured value which behaves much like an indexed array. Each element of a structured value is, in turn, a node that represents either an elementary value or a structured value. The pointers in the data tokens refer to one of these nodes and a reference count is maintained. A node which is no longer referred to, either directly or indirectly, in the graph is removed by an implicit garbage collector.

Dennis [1974] went on to show that implementation is possible without need of copying arbitrarily complex values. As Dennis's dataflow programs were always functional, it was necessary that modifying a value should result in a new value, without changing the original. Whenever an elementary value is modified, a new node is simply added to the heap. Whenever a structured value is modified, and there is more than one reference to the value, a new root node is added to the heap, pointing to the same values as the original root node, with the exception of the one value that was to have been modified for which a new node is created. This is illustrated by Figure 7, which shows the effect on Dennis's data heap when a value A, which represents the array [a, b, c], is modified. The second element is modified to create a new value B, which represents the array [a, e, c]. In the memory heap, value B retains references to all the data that is not modified, thus saving time by not copying the entire data structure. Meanwhile, value A remains unmodified, preserving the functional semantics of the model.

This method prevents the unbounded copying of arbitrarily complex values. It also permits the sharing of identical data elements which saves memory. However, it is not an ideal solution for all situations. For example, if the values in a 100-element array are being modified sequentially by a loop, this solution would require making 100 new data structures in the process, notwithstanding the fact that they are not copying the entire array each time. A good compiler could detect such a loop and prevent needless copying such as this. Excessive overhead in the Dennis approach was also examined and reported by Gajski et al. [1982].

A second problem, and one which became more evident as research progressed [Ackerman 1982], was that the use of data structures can reduce parallelism in a dataflow program due to the long delay between creating the structure, and all parts of it being completed. To use Ackerman's [1982] example, consider a dataflow program that has two main sections. The first creates a 100-element array and populates it, one element at a time. The second takes the array and reads the elements, one at a time. In this case, the second part cannot begin to execute until the first part is complete, even though it could be reading element 1 while the first part of the program is writing element 2, and so on. In this case, a program that could conceivably execute in 101 time units, takes 200 time units to complete. This delay proved to be frequently unnecessary, and led to the development of I-structures.

6.3.2. I-Structures. To overcome this problem, Arvind and Thomas [1980] proposed a system that they called *I-structures*. They observed that the problem with Dennis's approach was that it imposed too strict a control structure on the computations that filled in the components of the data structure [Arvind et al. 1989]. Ideally, what was needed was some way of allowing more flexible access to the data structures but which—crucially—would not destroy the functional semantics of dataflow.

I-structures are related to lazy evaluation. A data structure is created in memory, but its constituent fields are left blank. Each field can either store a value or be "undefined." A value can only be stored in a field that is currently undefined. Thus I-structures obey the single-assignment rule. Any attempt to read from an undefined field is deferred until that value becomes available, that is, until a value is assigned to it.

By following this set of rules, a data structure can be "transmitted" to the rest of the program as soon as it is created, while the sender continues to populate the fields of that structure. Meanwhile, the receiver can begin to read from the structure. Referring again to Ackerman's [1982] example, this would dramatically improve the performance of some programs. Although they are somewhat opposed to the purity of dataflow, I-Structures have been widely adopted [Arvind et al. 1988; Culler et al. 1995; Keller 1985].

However, while I-structures do solve problems of unnecessary delays in functional dataflow programs, they do not address the initial problem of copying data structures in order to modify them. Gajski et al. [1982] pointed out other issues related to the overhead of storing and fulfilling deferred reads in this approach, although Arvind and Culler [1986] argued that this overhead is small and easily outweighed by the benefits. This latter view appears to be supported by the experiments of Arvind et al. [1988].

6.3.3. Hybrid Structures. Hurson et al. [1989] examined both Dennis's copying strategy and the later I-structures. As discussed above, they found that the former had high-overhead issues and wasted potential parallelism, while the latter wasted space as it was unable to share common substructures. They claimed that their proposed hybrid structures carefully combined the advantages brought forth by both copying and sharing. To demonstrate their proposed structure, they used arrays, although the method can be applied to any form of data structure.

Their method represents each array as an array template (for full details, see Hurson et al. [1989]). The array template has a reference count and can either refer to an original array or a modified array. In the case of an original array, the template points to a sequential area of memory that contains the elements of the array. Whenever an element of the array is "modified," a new "modified" array template is created to represent the new array. It contains an area of memory that represents the new array, but with only the modified value filled in. The other values are marked as absent and a link is provided back to the original array. An attempt to read from the new array will either return a modified value or, if the value is not there, the link to the original array will be followed and the value retrieved from there.

If another value is modified in the new array, and its reference count remains 1, it can be modified in situ without having to create a new array template. In order to prevent copying the entire array each time, a new array is required; hybrid structures allow large arrays to be broken up into equally sized blocks, each storing a certain number of elements. Because the blocks are of identical size, looking for an element within them can be achieved in constant time. Blocks themselves have reference counts, allowing for sharing of subportions of arrays.

Experiments reported in the same paper [Hurson et al. 1989] suggest that hybrid structures lead to improvements in both performance and storage over the copying method of Dennis [1974] and I-structures, although it does contain a certain element of overhead.

While the three approaches outlined above, and their derivatives, have resolved many of the problems related to efficiently implementing data structures in the dataflow model, those problems nevertheless remain open issues. A brief overview of data structures and dataflow was given in Lee and Hurson [1993]. Efforts to reduce unnecessary copying and to reduce wasted memory from needless duplication, and the desire not to reduce parallelism when implementing data structures, remain topics for further research. Data structures will always sit uneasily within pure dataflow models, but as their provision is virtually essential, it is an issue that must necessarily be examined.

6.4. Nondeterminism

The deterministic nature of dataflow graphs has been promoted many times as an advantageous feature [Kahn 1974; Karp and Miller 1966; Kosinski 1973; Naggar et al. 1999; Verdoscia and Vaccaro 1998]. This is because the dataflow concept lends itself well to mathematical analysis and proofs [Kahn 1974] and nondeterminism would destroy or limit many essential properties. Weng [1975] observed that in von Neumann languages, concurrency constructs almost always introduce unwanted concurrency into programs, and that developing distributed systems in this model is made extremely difficult by this fact. Most dataflow programming languages are determinate, and the nondeterminacy in some of those that are not is not always intentional—often being the result of imperfect implementation decisions. Valid, Cajole, and DL1 have very limited nondeterminate features.

However, it is also widely accepted that there are many applications that actually require non-determinacy. These are systems that are essentially operating in nondeterminant environments, such as booking systems and database access systems. This was recognized early in the development of dataflow languages. Dennis [1974] conceded the point, and Kahn [1974], after his detailed mathematical analysis of determinate dataflow graphs, conceded that his model was severely limited because it could produce only determinate programs. While not dismissing the possibility of extending the theory to nondeterminate programs, the task appears daunting: Kahn [1974] remarked only that he did not think it was impossible, but did not find it obvious how to do it satisfactorily. This view was supported by Kosinski [1978], who reported that attempts to formalize nondeterminate dataflow graphs had been rather unsatisfactory due to their complexity.

The dichotomy in regard to nondeterminism appears to be the result of a division between those who wish to use dataflow as a means to ease the formal proof and analysis of programs and those who wish to use dataflow for all varieties of programming problems. The former regard determinacy as essential, whereas the latter regard the provision of nondeterminacy as essential.

This problem can be resolved by providing well-structured nondeterminacy. In admitting the need for nondeterminacy, Dennis [1974] nevertheless insisted that he wanted to be able to guarantee users of his language that his program was determinate if they desired such a guarantee. Arvind et al. [1977] proposed that nondeterminacy be permitted only by very explicit means, to provide it for those who want it, but guarantee its absence, if not. They demonstrated two constructs: the dataflow monitor and the nondeterministic merge as vehicles for this.

The nondeterminate merge appears to be able to solve many of the problems associated with the lack of nondeterminacy. Semantically, it is a node that takes two input arcs and one output arc and merges the two streams in a completely arbitrary way. In most cases, such as a booking system, this is all the nondeterminacy that is required. The advantage of this is that the nondeterminism can be readily identified. It is even possible to have nondeterminate subgraphs within a graph that is otherwise determinate. Therefore, it may be possible to apply mathematical principles to the graph even if it does have nondeterminate sections.

If dataflow is to become an acceptable basis for general-usage programming languages, nondeterminacy is essential. As well as having disadvantages for formal proofs, nondeterminacy also damages the software engineering process by making debugging more difficult. Therefore, the question is how to successfully control the propagation of nondeterminacy in dataflow systems, but still permit the software engineer to write usable programs.

7. CONCLUSION

In this article, the history of dataflow programming has been charted to the present day. Beginning with the theoretical foundations of dataflow, the design and implementation of fine-grained dataflow hardware architectures have been explored. The growing requirement for dataflow programming languages was addressed by the creation of a functional paradigm of languages, and the most relevant of these have been discussed.

The discovery that fine-grained dataflow had inherent inefficiencies led to a period of decline in dataflow research in the 1980s and early 1990s. However, research in the field resumed in the 1990s with the acceptance that the best dataflow hardware techniques would come from merging dataflow and von Neumann techniques. This led to the development of hybrid architectures, whose primary trait was a move away from fine-grained parallelism toward more coarse-grained execution.

The most important development in dataflow programming languages in the 1990s was the advent of dataflow visual programming languages, DFVPLs, which have been explored. Integral to a DFVPL is its development environment, and these have been discussed. The change in motivation for pursuing DFVPLs toward software engineering has been noted. Finally, many issues remain open in dataflow programming, and four of these have been discussed.

Five key conclusions can be drawn regarding the current state of dataflow programming.

—The major change in dataflow research as a whole has been the move away from fine-grained parallelism towards medium- and coarse-grained parallelism.

—The major change in the past decade in dataflow programming has been the advent of dataflow visual programming languages.

—As it is visualization that is key to a visual programming language, the distinction between a dataflow visual programming language and its environment has become blurred and the two must now be treated as one unit.

—Dataflow languages increasingly deserve to be treated as coordination languages, an important area of research with the advent of heterogeneous distributed systems.

—The three key open issues in dataflow programming remain the representation of control-flow structures, the representation of data structures, and the visualization of execution.

REFERENCES

Ackerman, W. 1982. Data flow languages. IEEE *Comput. 15*, 2, 15–25.

Ackerman, W. B. and Dennis, J. B. 1979. VAL—A value-oriented algorithmic language: Preliminary reference manual. Tech Rep. 218. MIT, Cambridge, MA.

Amamiya, M., Hasegawa, R., and Ono, S. 1984. Valid, a high-level functional programming language for data flow machines. *Rev. Electric. Comm. Lab. 32*, 5, 793–802.

Arvind and Culler, D. E. 1983. The tagged token dataflow architecture (preliminary version). Tech. Rep. Laboratory for Computer Science, MIT, Cambridge, MA.

Arvind and Culler, D. E. 1986. Dataflow architectures. *Ann. Rev. Comput. Sci. 1*, 225–253.

Arvind, Culler, D. E., and Maa, G. K. 1988. Assessing the benefits of fine-grain parallelism in dataflow programs. *Int. J. Supercomput. Appl. 2*, 3, 10–36.

Arvind and Gostelow, K. P. 1977. Some relationships between asynchronous interpreters of a dataflow language. In *Proceedings of the IFIP WG2.2 Conference on the Formal Description of Programming Lanugages* (St. Andrews, Canada).

Arvind, Gostelow, K. P., and Plouffe, W. 1977. Indeterminancy, monitors, and dataflow. In

Proceedings of the Sixth Symposium on Operating System Principles, 159–169.

Arvind, Gostelow, K. P., and Plouffe, W. 1978. An asynchronous programming language and computing machine, Tech. Rep. TR 114a. University of California, Irvine, Irvine, CA.

Arvind, Nikhil, R. S., and Pingali, K. K. 1989. I-structures: Data structures for parallel computing. *ACM Trans. Program. Lang. Syst. 11*, 4, 598–632.

Arvind and Nikhil, R. S. 1990. Executing a program on the MIT tagged-token dataflow architecture. *IEEE Trans. Comput. 39*, 3, 300–318.

Arvind and Thomas, R. 1980. I-structures: An efficient data type for functional languages. Tech. Memo 178. Laboratory for Computer Science, MIT, Cambridge, MA.

Ashcroft, E. A. and Wadge, W. W. 1977. Lucid, a nonprocedural language with iteration. *Comm. ACM 20*, 7 (July), 519–526.

Ashcroft, E. A. and Wadge, W. W. 1980. Some common misconceptions about Lucid. *ACM SIGPLAN Not. 15*, 10, 15–26.

Auguston, M. and Delgado, A. 1997. Iterative constructs in the visual data flow language. *Proceedings of the IEEE Conference on Visual Languages* (VL'97, Capri, Italy). 152–159.

Backus, J. 1978. Can programming be liberated from the von Neumann style? A functional style and its algebra of programs. *Comm. ACM 21*, 8 (Aug.), 613–641.

Barahona, P. and Gurd, J. R. 1985, Simulated performance of the Manchester multi-ring dataflow machine. In *Proceedings of the 2nd ICPC* (Sept.). 419–424.

Baroth, E. and Hartsough, C. 1995. Visual programming in the real world. In *Visual Object-Oriented Programming: Concepts and Environments*. Prentice-Hall, Upper Saddle River, NJ, 21–42.

Bernini, M. and Mosconi, M. 1994. VIPERS: A data flow visual programming environment based on the Tcl language. In *Proceedings of the Workshop on Advanced Visual Interfaces*. 243–245.

Bhattacharyya, S. S. 1996. *Software Synthesis from Dataflow Graphs*. Kluwer Academic Publishers, Dordrecht, The Netherlands.

Bic, L. 1990. A process-oriented model for efficient execution of dataflow programs. *J. Parallel Distrib. Comput. 8*, 42–51.

Bic, L., Roy, J. M. A., and Nagel, M. 1995. Exploiting iteration-level parallelism in dataflow programs. In *Advanced Topics in Dataflow Computing and Multithreading*. IEEE Computer Society Press, Los Alamitos, CA, 167–186.

Bohm, W., Najjar, W. A., Shankar, B., and Roh, L. 1993. An evaluation of coarse grain dataflow code generation stretegies. In *Proceedings of the Working Conference on Massively Parallel Programming Models* (Berlin, Germany).

Buck, J. and Lee, E. A. 1995. The token flow model. In *Advanced Topics in Dataflow Computing and Multithreading*. IEEE Computer Society Press, Los Alamitos, CA, 267–290.

Burnett, M. M., Baker, M. J. et al. 1995. Scaling up visual programming languages. *IEEE Comput. 28*, 3, 45–54.

Comte, D., Durrieu, G., Gelly, O., Plas, A., and Syre, J. C. 1978. Parallelism, control and synchronisation expression in a single assignment language. *ACM SIGPLAN Not. 13*, 1, 25–33.

Cox, P., Giles, F., and Pietrzykowski, T. 1989. Prograph: A step towards liberating programming from textual conditioning. In *Procedings of the IEEE Workshop on Visual Languages*. 150–156.

Cox, P. and Smedley, T. 1996. A visual language for the design of structured graphical objects. In *Proceedings of the IEEE Symposium on Visual Languages*. 296–303.

Culler, D. E., Goldstein, S. C., Schauser, K. E., and von Eicker, T. 1995. Empirical study of a dataflow language on the CM-5. In *Advanced Topics in Dataflow Computing and Multithreading*. IEEE Computer Society Press, Los Alamitos, CA, 187–210.

Davis, A. L. 1974. Data driven nets—A class of maximally parallel, output-functional program schemata. Tech. Rep. IRC Report. Burroughs, San Diego, CA.

Davis, A. L. 1978. The architecture and system method of DDM1: A recursively structured data driven machine. In *Proceedings of the 5th Annual Symposion on Computer Architecture* (New York). 210–215.

Davis, A. L. 1979. DDN's—a low level program schema for fully distributed systems. In *Proceedings of the 1st European Conference on Parallel and Distributed Systems* (Toulouse, France). 1–7.

Davis, A. L. and Keller, R. M. 1982. Data flow program graphs. *IEEE Comput. 15*, 2, 26–41.

Davis, A. L. and Lowder, S. A. 1981. A Sample management application program in a graphical data-driven programming language. In *Digest of Papers Compcon Spring, February 1981*. 162–165.

De Jong, M. D., and Hankin, C. L. 1982. Structured data-flow programming. *ACM SIGPLAN Not. 17*, 8, 18–27.

Dennis, J. 1977. *A Language Design for Structured Concurrency. The Design and Implementation of Programming Languages*. Lecture Notes in Computer Science, vol. 54. Springer-Verlag, Berlin, Germany, 23–42.

Dennis, J. B. 1974. First version of a data flow procedure language. In *Proceedings of the Symposium on Programming* (Institut de Programmation, University of Paris, Paris, France). 241–271.

Dennis, J. B. 1980. Data flow supercomputers. *IEEE Comput. 13*, 11 (Nov.), 48–56.

DENNIS, J. B. AND MISUNAS, D. P. 1975. A preliminary architecture for a basic data-flow processor. In *Proceedings of the Second Annual Symposium on Computer Architecture*. 126–132.

GAJSKI, D. D., PADUA, D. A., KUCLE, D. J., AND KUH, R. H. 1982. A second opinion on data-flow machines and languages. *IEEE Comput. 15*, 2, 58–69.

GAO, G. R. AND PARASKEVAS, Z. 1989. Compiling for dataflow software pipelining. In *Proceedings of the Second Workshop on Languages and Compilers for Parallel Computing*.

GELERNTER, D. AND CARRIERO, N. 1992. Coordination languages and their significance. *Comm. ACM 35*, 2, 97–107.

GELLY, O. 1976. LAU software system: A high-level data-driven language for parallel processing. In *Proceedings of the International Conference on Parallel Processing* (New York, NY).

GHITTORI, E., MOSCONI, M., AND PORTA, M. 1998. Designing and testing new programming constructs in a data flow VL. Tech. Rep. Università di Pavia, Pavia, Italy.

GLAUERT, J. R. W. 1978. A single assignment language for data flow computing, Master's thesis. University of Manchester, Manchester, U.K.

GREEN, T. R. G. AND PETRE, M. 1996. Usability analysis of visual programming environments: A "Cognitive Dimensions" Framework. *J. Vis. Lang. Comput. 7*, 131–174.

GURD, J. R. AND BOHM, W. 1987. Implicit parallel processing: SISAL on the Manchester dataflow computer. In *Proceedings of the IBM-Europe Institute on Parallel Processing* (Aug., Oberlech, Austria).

HANKIN, C. L. AND GLASER, H. W. 1981. The dataflow programming language CAJOLE—an informal introduction. *ACM SIGPLAN Not. 16*, 7, 35–44.

HARVEY, N. AND MORRIS, J. 1993. NL: A general purpose visual dataflow language, Tech. Rep. University of Tasmania, Tasmania, Australia.

HARVEY, N. AND MORRIS, J. 1996. NL: A parallel programming visual language. Australian *Comput. J. 28*, 1, 2–12.

HELSEL, R. 1994. *Cutting Your Test Development Time with HP VEE*. Prentice-Hall, HP Professional Books, Englewood Cliffs, NJ.

HILS, D. D. 1992. Visual languages and computing survey: Data flow visual programming languages. *J. Vis. Lang. Comput. 3*, 1, 69–101.

HURSON, A. R., LEE, B., AND SHIRAZI, B. 1989. Hybrid structures: A scheme for handling data structures in a data flow environment. In *Proceedings of the Conference on Parallel Architectures and Languages* (PARLE), 323–340.

IANNUCCI, R. A. 1988. Towards a dataflow/von Neumann hybrid architecture. In *Proceedings of the ICSA-15* (Honolulu, HI). 131–140.

IWATA, M. AND TERADA, H. 1995. Multilateral diagrammatical specification environment based on data-driven paradigm. In *Advanced Topics in Dataflow Computing and Multithreading*. IEEE Computer Society Press, Los Alamitos, CA, 103–112.

JAGANNATHAN, R. 1995. Coarse-Grain Dataflow Programming of Conventional Parallel Computers. In *Advanced Topics in Dataflow Computing and Multithreading*. IEEE Computer Society Press, Los Alamitos, CA, 113–129.

KAHN, G. 1974. The semantics of a simple language for parallel programming. In *Proceedings of the IFIP Congress 74* (Amsterdam, The Netherlands). 471–475.

KARP, R. AND MILLER, R. 1966. Properties of a model for parallel computations: Determinacy, termination, queueing. *SIAM J. 14*, 1390–1411.

KELLER, R. M. 1985. Rediflow architecture prospectus, Tech. Rep. No. UUCS-85-105. Department of Computer Science, University of Utah, Salt Lake City, Utah.

KELLER, R. M. AND YEN, W. C. J. 1981. A graphical approach to software development using function graphs. In *Digest of Papers Compcon Spring, February 1981*. 156–161.

KIMURA, T. AND MCLAIN, P. 1986. Show and Tell user's manual. Tech. Rep. WUCS-86-4. Department of Computer Science, Washington University, St Louis, MO.

KIPER, J., HOWARD, E., AND AMES, C. 1997. Criteria for evaluation of visual programming languages. *J. Vis. Lang. Comput. 8*, 2, 175–192.

KOSINSKI, P. 1978. A straightforward denotational semantics for non-determinate data flow programs. In *Proceedings of the 5th ACM Symposium on Principles on Programming Languages*. ACM Press New York, NY.

KOSINSKI, P. R. 1973. A data flow language for operating systems programming. In 'Proceedings of ACM SIGPLAN-SIGOPS Interface Meeting.' *SIGPLAN Not. 8*, 9, 89–94.

LABVIEW. 2000. *Lab View User Manual*. National Instruments, Austin, TX.

LEE, B. AND HURSON, A. R. 1993. Issues in dataflow computing. *Adv. in Comput. 37*, 285–333.

LEE, B. AND HURSON, A. R. 1994. Dataflow architectures and multithreading. *IEEE Comput. 27*, 8 (Aug.), 27–39.

LEE, E. 1997. A denotational semantics for dataflow with firing. Memorandum UCB/ERL M97/3. Electronics Research Laboratory, University of California, Berkeley, Berkeley, CA.

LEE, E. AND MESSERSCHMITT, D. 1987. Static scheduling of synchronous dataflow programs for digital signal processing. *IEEE Trans. Comput. C-36*, 1, 24–35.

LEE, E. AND PARKS, T. 1995. Data-flow process networks. *Proc. IEEE. 83*, 5, 773–799.

MCGRAW, J., SKEDZIELEWSKI, S. ET AL. 1983. *SISAL—Streams and Iteration in a Single Assignment Language Reference Manual (Version 1.0)*. Livermore National Laboratory, Livermore, CA.

Morrison, J. P. 1994. *Flow-Based Programming: A New Approach to Application Development*. van Nostrand Reinhold, New York, NY.

Mosconi, M. and Porta, M. 2000. Iteration constructs in data-flow visual programming languages. *Comput. Lang. 26*, 2-4, 67–104.

Naggar, W., Lee, E., and Gao, G. R. 1999. Advances in the dataflow computational model. *Parallel Comput. 25*, 1907–1929.

Najjar, W. A., Roh, L., and Wim, A. 1994. An evaluation of medium-grain dataflow code. *Int. J. Parallel Program. 22*, 3, 209–242.

Nikhil, R. S. and Arvind 1989. Can dataflow subsume von Neumann computing? In *Proceedings of the ICSA-16* (Jerusalem, Israel). 262–272.

Ning, Q. and Gao, G. R. 1991. Loop storage optimization for Dataflow machines. ACAPS Tech. Memo 23. School of Computer Science, McGill University, Montreal, P. Q., Canada.

Papadopoulos, G. M. 1988. Implementation of a general purpose dataflow multiprocessor. Tech. Rep. TR432. Laboratory for Computer Science, MIT, Cambridge, MA.

Papadopoulos, G. M. and Traub, K. R. 1991. Multithreading: A revisionist view of dataflow architectures. MIT Memo CSG-330. MIT, Cambridge, MA.

Plaice, J. 1991. RLUCID, A General Real-Time Data-Flow Language. Lecture Notes in Computer Science, vol. 571. Springer-Verlag, Berlin, Germany, 363–374.

Rasure, J. and Williams, C. 1991. An integrated data flow visual language and software development environment. *J. Vis. Lang. Comput. 2*, 217–246.

Richardson, C. 1981. Manipulator control using a data-flow machine. Doctoral dissertation. University of Manchester, Manchester, U.K.

Sakai, S., Yamaguchi, Y., Hiraki, K., Kodama, Y., and Yuba, T. 1989. An architecture of a dataflow single chip processor. In *Proceedings of the 16th International Symposium on Computer Architecture*. 46–53.

Serot, J., Quenot, G., and Zavidovique, B. 1995. A visual dataflow programming environment for a real time parallel vision machine. *J. Vis. Lang. Comput. 6*, 4, 327–347.

Shizuki, B., Toyoda, M., Shibayama, E., and Takahashi, S. 2000. Smart browsing among multiple aspects of data-flow visual program execution, using visual patterns and multi-focus fisheye views. *J. Vis. Lang. Comput. 11*, 5, 529–548.

Shürr, A. 1997. BDL—a nondeterministic data flow programming language with backtracking. In *Proceedings of the IEEE Conference on Visual Languages* (VL'97, Capri, Italy).

Silc, J., Robic, B., and Ungerer, T. 1998. Asynchrony in parallel computing: from dataflow to multithreading. *Parallel Distrib. Comput. Pract. 1*, 1, 3–30.

Sterling, T., Kuehn, J., Thistle, M., and Anastasis, T. 1995. Studies on Optimal Task Granularity and Random Mapping. In *Advanced Topics in Dataflow Computing and Multithreading*. IEEE Computer Society Press, Los Alamitos, CA, 349–365.

Tanimoto, S. 1990. VIVA: A visual language for image processing. *J. Vis. Lang. Comput. 1*, 127–139.

Toyoda, M., Shizuki, B., Takahashi, S., Matsuoka, S., and Shibayama, E. 1997. Supporting design patterns in a visual parallel data-flow programming environment. In *Proceedings of the IEEE Symposion on Visual Languages* (Capri, Italy). 76–83.

Treleaven, P. C., Brownbridge, D. R., and Hopkins, R. P. 1982. Data-driven and demand-driven computer architecture. *ACM Comput. Surv. 14*, 1, 93–143.

Treleaven, P. C. and Lima, I. G. 1984. Future computers: Logic, data flow, . . . , control flow? *IEEE Comput. 17*, 3 (Mar.), 47–58.

Veen, A. H. 1986. Data flow machine architecture. ACM *Comput. Surv. 18*, 4, 365–396.

Verdoscia, L. and Vaccaro, R. 1998. A high-level dataflow system. *Comput. J. 60*, 4, 285–305.

Wadge, W. W. and Ashcroft, E. A. 1985. *Lucid, the Dataflow Programming Language*. APIC Studies in Data Processing, no. 22. Academic Press, New York, NY.

Wail, S. F. and Abramson, D. 1995. Can Dataflow Machines be Programmed with an Imperative Language? In *Advanced Topics in Dataflow Computing and Multithreading*. IEEE Computer Society Press, Los Alamitos, CA, 229–265.

Watson, I. and Gurd, J. R. 1979. A prototype data flow computer with token labelling. In *Proceedings of the National Computer Conference*. 623–628.

Weng, K. S. 1975. Stream oriented computation in recursive data-flow schemas. Tech. Rep. 68. Laboratory for Computer Science, MIT, Cambridge, MA.

Whiting, P. and Pascoe, R. 1994. A history of dataflow languages. *IEEE Ann. Hist. Comput. 16*, 4, 38–59.

Whitley, K. 1997. Visual programming languages and the empirical evidence for and against. *J. Vis. Lang. Comput. 8*, 1, 109–142.

Yu, Y. J. and D'hollander, E. H. 2001. Loop parallelization using the 3D iteration space visualizer. *J. Vis. Lang. Comput. 12*, 2, 163–181.

Received October 2001; revised September 2002, September 2003; accepted April 2004