# A Localized Condition Handling Construct for a Graphical Dataflow Language

David D. Langan, Ph. D.
University of South Alabama
Mobile, Alabama 36688

## ABSTRACT

This paper gives a brief description of dataflow programming and of the condition handling constructs used in existing dataflow languages. It is argued that existing mechanisms are deficient both in terms of flexibility and performance. A construct that provides for increased flexibility and improved performance is presented. The construct described here is called a supervisor and has three component parts called (a) the input acceptance test, (b) the condition handler and (c) the output acceptance test. The basic dataflow model is expanded to include condition arcs and tokens. The construct presented here is a part of a more comprehensive scheme for condition handling in dataflow models [Langan 88].

## AN INTRODUCTION TO DATAFLOW

The increased use of computers, the decrease in their cost, and the application of computers to more complex problems, has led to languages and architectures designed to take advantage of increased processing power. Languages have evolved that allow the programmer to identify computations that can be executed in asynchronous fashion (e.g., Ada and Modula-2). Such parallel processing languages allow for a clear statement of algorithms that require parallel execution.

The languages mentioned above all have one aspect in common; they are control flow based languages which assume multiple loci of execution. Another approach for the description of parallel computations is dataflow. The essential characteristics of basic dataflow are:

(1) A dataflow graph is collection of nodes, each providing a side effect free function, connected by directed arcs. The arcs are the data paths along which the data tokens flow from node to node.

(2) A node executes (or fires), consuming its input tokens, only if the input it requires is available and if adequate space is available for any output tokens.

(3) All communication including input values, output values and synchronization signals is via tokens.



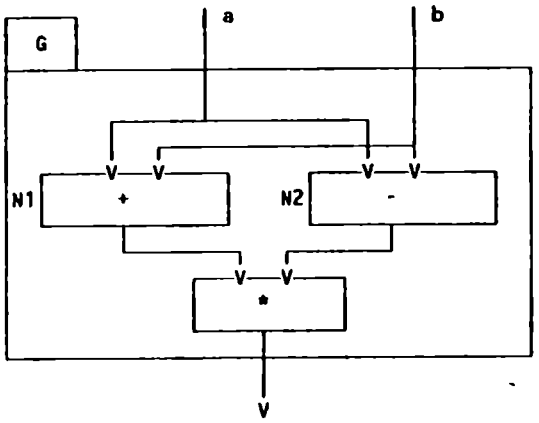


Figure 1: A Dataflow Graph

Figure 1 shows a dataflow graph called G that computes (a+b)*(a-b). The computations for + and - can be performed as soon as tokens holding values for a and b, appear on the input arcs to the nodes labeled N1 and N2. The + and - may be executed in parallel.

Dataflow languages differ from other parallel processing languages in that their execution is data driven as opposed to being driven by control flow. Many descriptions of dataflow have appeared in the literature of the past twenty years. [Karp 1966, Dennis 1975, Landry 1981, Veen 1986], define different dataflow models. These models differ with respect to the atomic operations available, the use of arcs, and the rules concerning the firing of nodes. As a result, a "standard" dataflow model does not exist; however, most of the models proposed are basically similar. These models assume that nodes represent deterministic side-effect free computations. An excellent overview of both proposed models and architectures may be found in [Arvind 1986, Srini 1986, Veen 1986].

## DEFINITION OF A CONDITION

A condition is an event that is deemed noteworthy. An event, at the hardware level, might be the existence of a designated state after a given instruction (e.g., overflow after multiplication) or it might refer to an attempt to make an invalid memory reference (e.g., range error on an array reference). At the programming level, an event might refer to a particular set of values for a specified set of variables during execution. Conditions may include states predefined by the system, language support environment, or the user of the programming language.

The existence of a noteworthy event may be independent of its detection. The act of checking to determine if a condition has occurred or that the condition exists, is referred to as condition detection. The signaling of the fact that a condition exists is called posting the condition. A condition handler is a collection of responses that are taken if a condition is detected. A condition handler may include the resumption, termination or modification of the execution environment of the associated code.

The major problems with conditions concern detection, flow of control, association of condition handlers with code, and the transfer of condition notification through environments.

## EXISTING DATAFLOW LANGUAGES

Proposals for condition handling in dataflow differ substantially. Three major efforts at creating a new high level dataflow language have been VAL [Ackerman 1979], Id [Arvind 1978] and Lucid [Wadge 1985]. In VAL the problem of handling conditions is dealt with by extending each token type to include condition values (e.g., OVERFLOW [Integer] or ZERO_DIVIDE [Real]). The semantics of each operator is extended to include the various condition values that might be received (e.g., POSITIVE_OVERFLOW + X = POSITIVE_OVERFLOW if 0 ≤ X, X = POSITIVE_OVERFLOW or X = POSITIVE_UNDERFLOW, otherwise UNKNOWN).

Plouffe [Plouffe 1980] proposes a mechanism for exception handling and recovery in applicative systems and demonstrates his proposal in terms of Id. Plouffe's mechanism differs from the VAL approach in that he creates a type specifically for conditions. This "condition" type permits more flexibility for condition handling. For example, the user could define new types of conditions, whereas with VAL the user is limited to predefined condition values. Plouffe's error type allows for a "composite" type of condition through the concatenation of error notifications.

In Lucid conditions are also treated as a separate data type, but the various operators provided have an extended semantics to allow for the use of condition tokens. Unlike the two approaches described above, the condition tokens do not carry with them any information that would convey the original cause of the condition and hence condition handling is made more difficult.

The three linguistic mechanisms proposed for the textual dataflow languages VAL, Id [Plouffe 1980] and Lucid demonstrate certain similarities; in particular, all three do the following:

> (1) extend the basic operators to include the facility to produce an output even if a condition should occur,
>
> (2) extend functionality to deal with condition values,
>
> (3) allow conditions to flow through the graph until a point is reached where some action may be taken.

The proposals differ, however, in terms of how they treat the association of condition handlers. Plouffe demonstrates how condition handlers can be associated with expressions in Id. He shows how they can be used to implement either a forward (i.e., use current state to respond to the condition) or backward (i.e., restore a "correct" earlier state) error recovery scheme. By contrast, the VAL and Lucid proposals do not address this problem.

All three approaches have several shortcomings. Each was designed especially for textual dataflow languages. While their solutions are applicable to graphic dataflow languages, the behavioral properties they exhibit are not entirely satisfactory. In particular, a condition at one point in the execution must flow through the graph taking additional time and allowing useless computations to be performed.

## THE PROBLEM

Dataflow languages and models have been used by researchers for a wide variety of tasks such as for complex weather simulation problems [Dennis 1984], and for signal processing in real time systems [Hartimo 1986]. If dataflow is to emerge as a viable tool for general use, it must be augmented to include provisions for resource sharing and condition handling.

Responding to conditions in an asynchronous execution environment is difficult due to the complexity of state information for such computations and due to the totally independent (potentially distributed) nature of the execution itself. Similarly, the task of attempting to debug programs written to execute in such an environment is difficult because the problems might be related to timing of the execution. Condition handling and debugging are related problems. As a partial solution to this problem an extension to the dataflow model is presented here.

The semantics of the enhanced model are

presented using an extended graphical dataflow language for an operational definition. The operational definition identifies the dataflow system support required to provide the semantics of the enhanced model.

## EXTENSIONS FOR CONDITION HANDLING

The major extension presented here includes the addition of "conditions" to the model. This addition includes condition tokens, condition arcs, system and user posted conditions, and association of condition detection and handling with any node.

Each node has associated with it the name of the operation to be performed and a firing rule that identifies:

(1) inputs required for the firing to begin,

(2) output arcs that are supposed to receive a token as a result of the firing,

(3) if desired, a limit on execution time.

Implicitly, those outputs not listed by (2) would be considered as optional for the given firing. In the above list, item (1) constitutes an input specification, (2) defines the output specification, and (3) is the associated execution specification. A node's firing rule is checked whenever there is the possibility that the node may fire. This includes the arrival of input tokens or the completion of a given execution (i.e., due to queued input tokens the node may have the inputs that it requires to fire again). The output portion of the firing rule is checked after the execution to confirm proper behavior.

The nodes in this dataflow model differ from nodes as used in other dataflow models in the following ways:

(1) The firing rules will include an output specification identifying required and optional outputs.

(2) In addition to the output values produced as a result of a node firing, a node may communicate information via the posting of a condition. Condition tokens and their role in a dataflow model are discussed in the next section.

(3) Node execution may be terminated if it is determined that the computation is not needed. This may be done if the time limit is exceeded or if a procedure wide decision to terminate has been made.

## CONDITIONS

Each of the functions available to a programmer must have its semantics adequately described for proper use of the function. The semantics of the operation to be performed must be described in terms of the inputs used and the outputs produced. A portion of the semantic description of a function must include those conditions that may be posted. For each condition that may be posted, there must be a clear statement of its meaning.

## FUNCTION POSTED CONDITIONS

The conditions that may be posted as a result of using a given function are called function posted conditions. These conditions are those detected and posted by the entity used to provide the node's functionality. This might be either an atomic operation or a dataflow procedure. A dataflow procedure is a dataflow graph where the arrival of the input tokens is synchronized and the departure of output tokens is synchronized. An atomic operation is one that is indivisible with respect to the dataflow model (e.g., an operation provided by the hardware or by a sequentially executed piece of code). The term "atomic function" may have a connotation implying a low level operation. This certainly need not be the case.

Conditions that are detected and posted by the dataflow system are called system posted conditions. For any node, the set of conditions that may be posted as a result of its execution is the union of the function posted and system posted conditions.

## SYSTEM POSTED CONDITIONS

In the case of a timing constraint specified as a part of the firing rule, the underlying system provides for the watchdog capability to monitor the elapsed time and to terminate the execution of the node. This capability permits the dataflow system to impose some resource usage constraints. If the node's execution is terminated due to the specified time limit being exceeded, this condition is posted to the program via a condition token placed on the node's condition arc. This condition is a system posted condition (TIMED OUT) and would carry with it adequate information for debugging purposes. The TIMED OUT condition is posted if a node exceeds the minimum of (a) the user specified time limit and (b) a system wide time limit. The user need not specify any time limit in the firing rule in which case only the system limit is enforced.

If the firing rule includes an output specification, then the system can mechanically verify that the output specification has been met. The system condition INADEQUATE OUTPUTS, is used to indicate that an output that had been explicitly identified as being required, was not produced.

Output specification verification requires system support and time. This overhead is optional in the sense that the default output specification for any firing rule is that all outputs are optional. The above system posted conditions reflect only those that are related to the behavior of a single node. Other conditions related to the behavior of an entire dataflow procedure as an entity can be defined if the model is further extended to include a procedure wide supervisor (see abstract entitled "Condition Handlers for Dataflow Procedures").

## CONDITION TOKENS AND CONDITION ARCS

The condition token is used in the posting procedure to carry the information concerning the condition from the point of posting to the point of condition handling. The token itself needs to convey in some fashion which conditions are being posted.

The maximum "size" of a condition token is known since the system posted and function posted conditions for a given node can be determined statically. This limits the size of the composite list of condition names that may be generated. To simplify graphs showing dataflow programs the condition arc emanating from each node in the graph is always displayed as the rightmost output arc from a node.

For debugging purposes, it may be desirable to have information in each condition token that identifies the node that posted the condition. The system itself should support an additional field in the condition token to identify the node. The unique name for each node may be user supplied or may be generated by the system. The name for the node that posted the condition is, however, known only within the dataflow procedure that contains it. An example of this scheme may be seen in viewing Figures 2a and 2b.

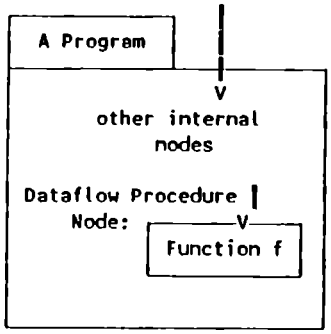


Figure 2a: Dataflow Program

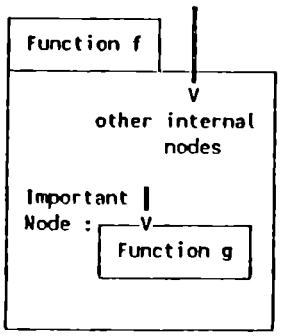


Figure 2b: Dataflow Procedure

In Figure 2b the name attached to any condition token posted from the node called "Important Node" is simply "Important Node". If, subsequently, a condition is posted from Function f, then the condition token within "A Program" (Figure 2a) from the node called "Dataflow Procedure Node" is called "Dataflow Procedure Node". This simple approach has a conceptual advantage as well as an implementation advantage. At the conceptual level, we are assured that a desirable level of information hiding is being supported. As procedure boundaries are crossed, the identification of the node that may have actually detected the "original condition" is lost. As an implementation issue, this approach is desirable because it guarantees that the node identifications being attached by the system are of a limited length. (Such might not be the case if full context names were being generated).

The naming of the node that posts a condition is of no importance in the case of a condition handler associated with a single node. The importance of this facility is more clearly seen when procedure condition handlers are presented.

## STATE INFORMATION WITH A CONDITION

In posting a given condition, it is often advantageous to include additional data pertaining to the precise nature of the condition or the environment within which it was detected (i.e., state information). A precise description of this additional information must accompany the description of the condition itself, so that a programmer writing a condition handler can properly use the extra information to respond to the condition. This specification is also important to an implementor of the model if the approach used requires a type specification for each token. The use of additional information does not include the name of the node that posted the condition as that would be provided by the system support. The information contained in a condition token includes:

(1) Source Node Identification (System Supplied)

(2) List of :
  (A) Condition Name
  (B) Additional information associated with the specific condition.

## NODE SUPERVISORS

An input acceptance test (IAT) is a test that is applied to one or more of the inputs to the node in order to test the acceptability of the values of the input tokens. An output acceptance test (OAT) is a test that is applied to the outputs produced as the result of the firing of a node. A node supervisor is a named collection of the input acceptance test, the output

acceptance test and the condition handler (CH), that may be associated with a specified node. A node supervisor need not contain all of the components mentioned above. It may, for example, consist of only an input acceptance test and a condition handler. The association of a supervisor with a node forms an extended node.

## INPUT ACCEPTANCE TEST

The input acceptance test (IAT) may consist of several sub-components if the requirements on individual inputs are considered separately. The input acceptance test is a form of condition detection. The choices of action available within a supervisor are somewhat limited due to the requirement of synchronization on outputs from the extended node. After the supervisor detects an input-value related condition it may (a) pass a set of input values (possibly modified) to the node to be fired, or (b) select to bypass the firing of the node and simply produce the outputs for the node. In the latter case, the input acceptance test may also post a condition for the node. In the former case, a condition may be posted, but only at the time of termination as all outputs (including the condition) are synchronized. The input acceptance test is involved in condition detection, condition handling and in the posting procedure.

The role of the input acceptance test is graphically represented Figure 3. In this and subsequent figures, solid lines represent "data arcs" and the dotted lines indicate condition arcs. The single solid lines represent a single arc while the double lines are used to represent one or more arcs. The distinction between data and conditions is made primarily as a convenience to emphasize the route of posted conditions. The condition tokens are otherwise identical to the data tokens.

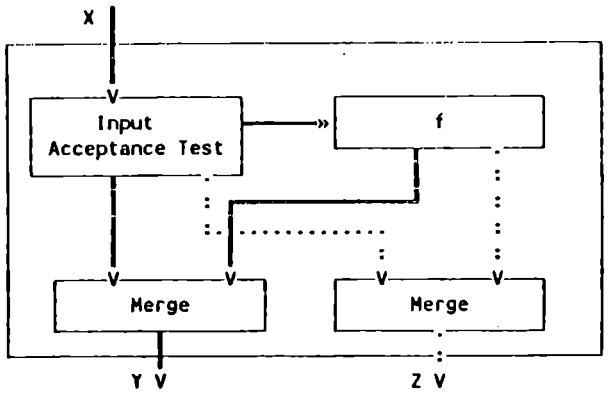


Figure 3: An Extended Node

A node supervisor is itself a named entity. If the node N1 in Figure 4a is to have the supervisor "S" associated with it, then the extended node, EN1, is graphically represented by the diagram in Figure 4b.

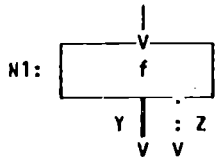


Figure 4a : Node

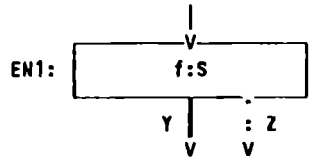


Figure 4b : Extended Node

## OUTPUT ACCEPTANCE TEST AND CONDITION HANDLER

In the case of a single node, the output of the computation is synchronized, i.e., the outputs and conditions are available at the same time. For this reason, the output acceptance test (OAT) and the condition handler (CH) are somewhat indistinguishable though they play different roles. Either or both of them might not be included in a node supervisor. When the node has finished its computation, the supervisor must determine what course of action is to be taken. Two options available to either of them include: retrying the node with either the same or modified input values, or producing a set of output values (with or without posting a condition). Figure 5 uses a state transition diagram to show the stages of execution of the extended node.

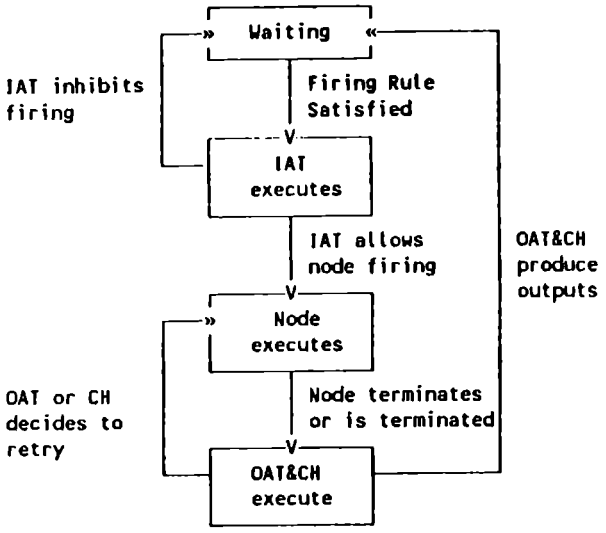


Figure 5 : Execution of Extended Node

A dataflow representation of the role of a node supervisor is shown in Figure 6. Figure 6 includes an input acceptance test (IAT). If the input acceptance test determines that the node need not be executed, it may produce output values (labeled as C) with or without posting a condition (labeled as D). If the node is to be fired, then it may be necessary to pass information from the IAT to the OAT&CH (arc labeled B) in addition to passing the input tokens (arc labeled E) on to the node. The execution of the function f may lead to the production of output tokens (arc labeled J) with or without a condition being posted (arc labeled K).

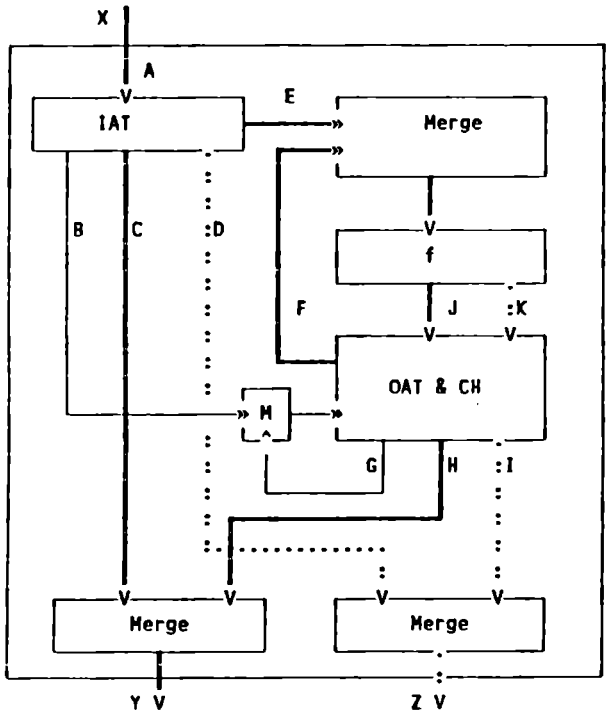


Figure 6 : A Fully Supervised Node

The OAT&CH executes after the node has terminated (or been terminated) and determines the course of action to take. It may choose to refire the node by passing it a set of input tokens (arc labeled F), in which case it may also need to pass some additional state information back to itself (arc labeled G), e.g. a retry count, so that an infinite loop of retries is avoided. The OAT&CH may also choose to simply produce the output tokens (arc labeled H) with or without posting a condition (arc labeled I).

## NAMING CONVENTION FOR NODE SUPERVISORS

The node supervisor treats the execution of the function it supervises in an indivisible fashion. The supervisor construct allows the programmer a convenient means of detecting conditions related to the input or output values and responding at a local level if a recovery action should be taken. The actions available at this level are limited to the reuse of the function, the production of some output value(s), or the posting of appropriate conditions to a more global level.

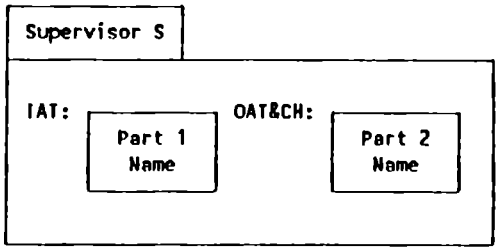


Figure 7 : Components of a Named Supervisor

The process of creating a supervisor entails providing a named collection containing the two components. We require a special naming convention for the creation of supervisors as shown in Figure 7.

## SYSTEM SUPPORT FOR NODES

A node is tightly bound to the supervisor associated with it. This binding requires intervention by the system support (e.g., the system may detect a condition related to the node and post the condition to the associated condition handler). To illustrate the tight binding between the node and its supervisor and to include the possibility of participation by the system support, a system support diagram is given in Figure 8. The flow of information is labeled according to Figure 6. In Figure 8, N1 is a node with a supervisor called S.

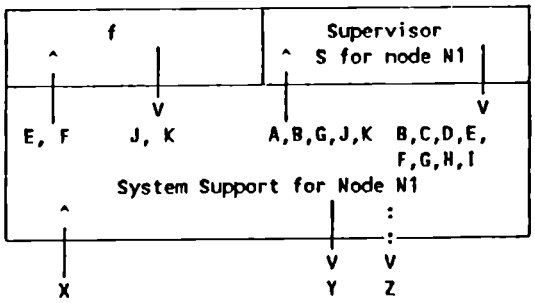


Figure 8 : Node System Support

The system support:

(1) maintains all token queues related to the extended node. For example, at the input boundary of the extended node (labeled X) it acts to queue tokens for a subsequent use. In this capacity it may be involved in communication with other parts of the overall system support (e.g., it may "respond" to inquiries regarding queue space on a given "arc". Such inquiries come from node system supports that want to pass an output token to this node).

(2) determines if the firing rule has been met and if so, initiates computation (labeled A).

(3) assists in the verification that the function f performed properly. This includes the detection of system posted conditions related to f (labeled J, C, H).

(4) initiates whatever actions must be performed to terminate the execution of the extended node if such a request should be passed to it from a more global level.

In Figure 8 the input to the extended node, labeled X, is shown as arriving "from below". This X is sent to the system support for node N1 from the procedure wide system support as part of the token routing activity.

## SUMMARY

This paper has presented a proposal for an extension to the dataflow model. The extension described include: (a) the addition of condition tokens and arcs, (b) system support for the detection and posting of system related conditions, and (c) the inclusion of a linguistic mechanism called a node supervisor to assist in the detection of input or output related conditions and handling of conditions posted by the node or system. This paper did not address how conditions are to be handled at the more global level. That topic is an extension of this work described in "Condition Handlers for Dataflow Procedures".

## ACKNOWLEDGEMENTS

The author wishes to thank Dr. Bruce Shriver and Dr. Steve Landry for their help during this research.

## REFERENCES

[Ackerman 1979] Ackerman, W. B. and Dennis, J. B., "VAL -- A Value-Oriented Algorithmic Language, Preliminary Reference Manual", MIT, Cambridge, Massachusetts, February 8, 1979.

[Arvind 1978] Arvind, Gostelow, and Plouffe, "The Id Report: An Asynchronous Programming language and Computing Machine", TR 114, Dept. of Computer Science, U. C. Irvine, California, Sept. 1978.

[Arvind 1986] Arvind and Culler, D. E., "Dataflow Architectures", MIT/LCS/TM-294, February 1986.

[Dennis 1975] Dennis, J and Misunas D., "A Preliminary Architecture for a Basic Data-Flow Processor", Proceedings of the Second Annual Symposium on Computer Architecture, IEEE, N.Y., 1975, pp. 126-132.

[Dennis 1984] Dennis, J. B., Gao, G. R., and Todd, K. W., "Modeling the Weather with a Data Flow Super Computer", IEEE Transactions on Computers, Vol. C-33, No. 7, July 1984, pp. 592 - 603.

[Hartimo 1986] Hartimo, I., Kronlof, K., Simula, O., and Skytta, J., "DFSP: A Data Flow Signal Processor", IEEE Transactions on Computers, Vol. C-36, No. 1, January 1986, pp. 23 - 32.

[Karp 1966] Karp, R. M. and Miller, R. E., "Properties of a Model for Parallel Computations: Determinacy, Termination, Queueing", SIAM Journal of Applied Mathematics, Vol. 14, November 1966.

[Landry 1981] Landry, S. P., "System Oriented Extensions to Dataflow", Ph. D. Thesis, Department of Computer Science, Univ. of Southwestern Louisiana, Lafayette, Louisiana, May 1981.

[Langan 1988] Langan, D., "A Dataflow Model Incorporating Condition Handling and Fault Tolerance", Ph. D. Thesis, Dept. of Computer Science, Univ. of Southwestern Louisiana, Lafayette, LA, May 1988.

[Plouffe 1980] Plouffe, W., "Exception Handling and Recovery in Applicative Systems", Ph. D. Thesis, Department of Computer Science, University of California, Irvine, California, 1980.

[Srini 1986] Srini, V. P., "An Architectural Comparison of Dataflow Systems", Computer, Vol. 19, No. 9, March 1986, pp. 68-88.

[Veen 1986] Veen, A. H., "Dataflow Machine Architecture", ACM Computing Surveys, Vol. 18, No. 4, December 1986, pp. 365 - 396.

[Wadge 1985] Wadge, W. W. and Ashcroft, E. A., Lucid, the Dataflow Programming Language, Academic Press, London, ISBN 0-12-729650-5, copyright 1985.