# Flow-based programming

From Wikipedia, the free encyclopedia

In computer programming, **flow-based programming** (**FBP**) is a programming paradigm that defines applications as networks of "black box" processes, which exchange data across predefined connections by message passing, where the connections are specified *externally* to the processes. These black box processes can be reconnected endlessly to form different applications without having to be changed internally. FBP is thus naturally component-oriented.

FBP is a particular form of dataflow programming based on bounded buffers, information packets with defined lifetimes, named ports, and separate definition of connections.

## Contents

## Introduction

Flow-based programming defines applications using the metaphor of a "data factory". It views an application not as a single, sequential process, which starts at a point in time, and then does one thing at a time until it is finished, but as a network of asynchronous processes communicating by means of streams of structured data chunks, called "information packets" (IPs). In this view, the focus is on the application data and the transformations applied to it to produce the desired outputs. The network is defined externally to the processes, as a list of connections which is interpreted by a piece of software, usually called the "scheduler".

The processes communicate by means of fixed-capacity connections. A connection is attached to a process by means of a port, which has a name agreed upon between the process code and the network definition. More than one process can execute the same piece of code. At any point in time, a given IP can only be "owned" by a single process, or be in transit between two processes. Ports may either be simple, or array-type, as used e.g. for the input port of the Collate component described below. It is the combination of ports with asynchronous processes that allows many long-running primitive functions of data processing, such as Sort, Merge, Summarize, etc., to be supported in the form of software black boxes.

Because FBP processes can continue executing as long they have data to work on and somewhere to put their output, FBP applications generally run in less elapsed time than conventional programs, and make optimal use of all the processors on a machine, with no special programming required to achieve this.[1]

The network definition is usually diagrammatic, and is converted into a connection list in some lower-level language or notation. FBP is often a visual programming language at this level. More complex network definitions have a hierarchical structure, being built up from subnets with "sticky" connections. Many other flow-based languages/runtimes are built around more traditional programming languages, the most notable example is RaftLib which uses C++ iostream-like operators to specify the flow graph.

FBP has much in common with the Linda[2] language in that it is, in Gelernter and Carriero's terminology, a "coordination language":[3] it is essentially language-independent. Indeed, given a scheduler written in a sufficiently low-level language, components written in different languages can be linked together in a single network. FBP thus lends itself to the concept of domain-specific languages or "mini-languages".

FBP exhibits "data coupling", described in the article on coupling as the loosest type of coupling between components. The concept of loose coupling is in turn related to that of service-oriented architectures, and FBP fits a number of the criteria for such an architecture, albeit at a more fine-grained level than most examples of this architecture.

FBP promotes high-level, functional style of specifications that simplify reasoning about system behavior. An example of this is the distributed data flow model for constructively specifying and analyzing the semantics of distributed multi-party protocols.

## History

Flow-Based Programming was invented by J. Paul Morrison in the early 1970s, and initially implemented in software for a Canadian bank.[4] FBP at its inception was strongly influenced by some IBM simulation languages of the period, in particular GPSS, but its roots go all the way back to Conway's seminal paper on what he called coroutines.[5]

FBP has undergone a number of name changes over the years: the original implementation was called AMPS (Advanced Modular Processing System). One large application in Canada went live in 1975, and, as of 2013, has been in continuous production use, running daily, for almost 40 years. Because IBM considered the ideas behind FBP "too much like a law of nature" to be patentable they instead put the basic concepts of FBP into the public domain, by means of a Technical Disclosure Bulletin, "Data Responsive Modular, Interleaved Task

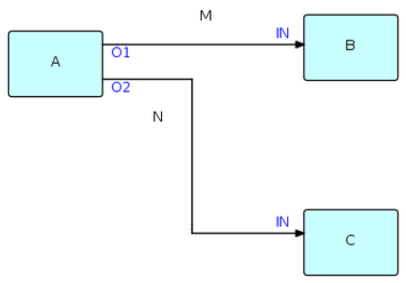
Programming System",[6] in 1971.[4] An article describing its concepts and experience using it was published in 1978 in the IBM Research IBM Systems Journal under the name DSLM.[7] A second implementation was done as a joint project of IBM Canada and IBM Japan, under the name "Data Flow Development Manager" (DFDM), and was briefly marketed in Japan in the late '80s under the name "Data Flow Programming Manager".

Generally the concepts were referred to within IBM as "Data Flow", but this term was felt to be too general, and eventually the name flow-based programming was adopted.

From the early '80s to 1993 J. Paul Morrison and IBM architect Wayne Stevens refined and promoted the concepts behind FBP. Stevens wrote several articles describing and supporting the FBP concept, and included material about it in several of his books.[8][9][10]. In 1994 Morrison published a book describing FBP, and providing empirical evidence that FBP led to reduced development times.[11]

## Concepts

The following diagram shows the major entities of an FBP diagram (apart from the Information Packets). Such a diagram can be converted directly into a list of connections, which can then be executed by an appropriate engine (software or hardware).



Simple FBP diagram

A, B and C are processes executing code components. O1, O2, and the two INs are ports connecting the connections M and N to their respective processes. It is permitted for processes B and C to be executing the same code, so each process must have its own set of working storage, control blocks, etc. Whether or not they do share code, B and C are free to use the same port names, as port names only have meaning within the components referencing them (and at the network level, of course).

M and N are what are often referred to as "bounded buffers", and have a fixed capacity in terms of the number of IPs that they can hold at any point in time.

The concept of *ports* is what allows the same component to be used at more than one place in the network. In combination with a parametrization ability, called Initial Information Packets (IIPs), ports provide FBP with a component reuse ability, making FBP a component-based architecture. FBP thus exhibits what Raoul de Campo and Nate Edwards of IBM Research have termed configurable modularity.

Information Packets or IPs are allocated in what might be called "IP space" (just as Linda's tuples are allocated in "tuple space"), and have a well-defined lifetime until they are disposed of and their space is reclaimed - in FBP this must be an explicit action on the part of an owning process. IPs traveling across a given connection (actually it is their "handles" that travel) constitute a "stream", which is generated and consumed asynchronously - this concept thus has similarities to the lazy cons concept described in the 1976 article by Friedman and Wise.[12]

IPs are usually structured chunks of data - some IPs, however, may not contain any real data, but are used simply as signals. An example of this is "bracket IPs", which can be used to group data IPs into sequential patterns within a stream, called "substreams". Substreams may in turn be nested. IPs may also be chained together to form "IP trees", which travel through the network as single objects.

The system of connections and processes described above can be "ramified" to any size. During the development of an application, monitoring processes may be added between pairs of processes, processes may be "exploded" to subnets, or simulations of processes may be replaced by the real process logic. FBP therefore lends itself to rapid prototyping.

This is really an assembly line image of data processing: the IPs travelling through a network of processes may be thought of as widgets travelling from station to station in an assembly line. "Machines" may easily be reconnected, taken off line for repair, replaced, and so on. Oddly enough, this image is very similar to that of unit record equipment that was used to process data before the days of computers, except that decks of cards had to be hand-carried from one machine to another.

Implementations of FBP may be non-preemptive or preemptive - the earlier implementations tended to be non-preemptive (mainframe and C language), whereas the latest Java implementation (see below) uses Java Thread class and is preemptive.

## Examples

### "Telegram Problem"

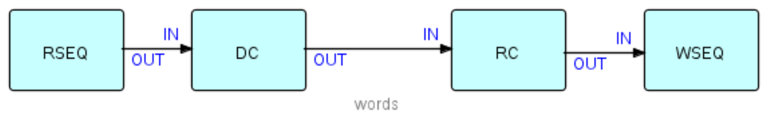
FBP components often form complementary pairs. This example uses two such pairs. The problem described seems very simple as described in words, but in fact is surprisingly difficult to accomplish using conventional procedural logic. The task, called the "Telegram Problem", originally described by Peter Naur, is to write a program which accepts lines of text and generates output lines containing as many words as possible, where the number of characters in each line does not exceed a certain length. The words may not be split and we assume no word is longer than the size of the output lines. This is analogous to the word-wrapping problem in text editors.[13]

In conventional logic, the programmer rapidly discovers that neither the input nor the output structures can be used to drive the call hierarchy of control flow. In FBP, on the other hand, the problem description itself suggests a solution:

- "words" are mentioned explicitly in the description of the problem, so it is reasonable for the designer to treat words as information packets (IPs)

- in FBP there is no single call hierarchy, so the programmer is not tempted to force a sub-pattern of the solution to be the top level.

Here is the most natural solution in FBP (there is no single "correct" solution in FBP, but this seems like a natural fit):



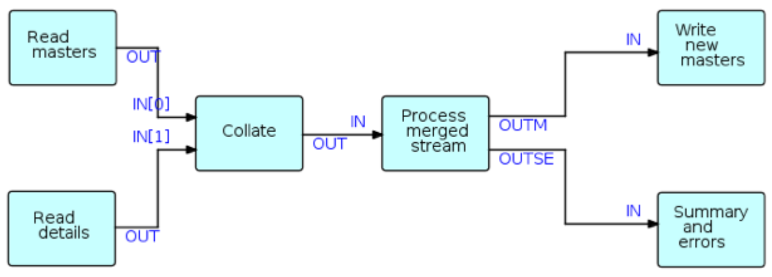
Peter Naur's "Telegram problem"

where DC and RC stand for "DeCompose" and "ReCompose", respectively.

As mentioned above, Initial Information Packets (IIPs) can be used to specify parametric information such as the desired output record length (required by the rightmost two components), or file names. IIPs are data chunks associated with a port in the network definition which become "normal" IPs when a "receive" is issued for the relevant port.

## Batch update

This type of program involves passing a file of "details" (changes, adds and deletes) against a "master file", and producing (at least) an updated master file, and one or more reports. Update programs are generally quite hard to code using synchronous, procedural code, as two (sometimes more) input streams have to be kept synchronized, even though there may be masters without corresponding details, or vice versa.



Canonical "batch update" structure

In FBP, a reusable component (Collate), based on the unit record idea of a Collator, makes writing this type of application much easier as Collate merges the two streams and inserts bracket IPs to indicate grouping levels, significantly simplifying the downstream logic. Suppose that one stream ("masters" in this case) consists of IPs with key values of 1, 2 and 3, and the second stream IPs ("details") have key values of 11, 12, 21, 31, 32, 33 and 41, where the first digit corresponds to the master key values. Using bracket characters to represent "bracket" IPs, the collated output stream will be as follows:

```
( m1 d11 d12 ) ( m2 d21 ) ( m3 d31 d32 d33 ) (d41)
```

As there was no master with a value of 4, the last group consists of a single detail (plus brackets).

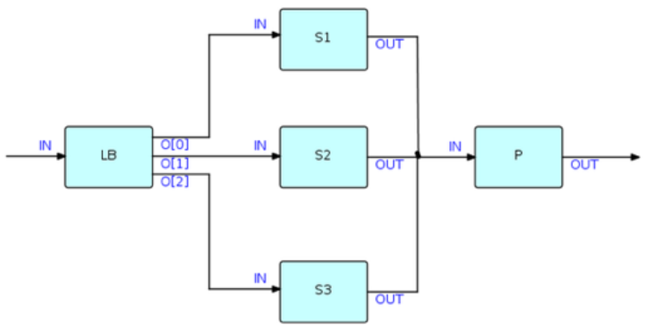
The structure of the above stream can be described succinctly using a BNF-like notation such as

```
{ ( [m] d* ) }*
```

Collate is a reusable black box which only needs to know where the control fields are in its incoming IPs (even this is not strictly necessary as transformer processes can be inserted upstream to place the control fields in standard locations), and can in fact be generalized to any number of input streams, and any depth of bracket nesting. Collate uses an array-type port for input, allowing a variable number of input streams.

## Multiplexing processes

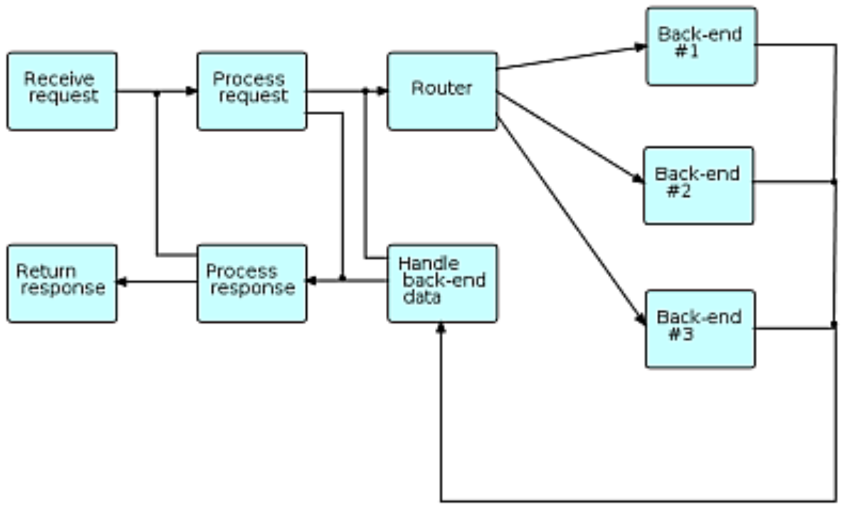
Flow-based programming supports process multiplexing in a very natural way. Since components are read-only, any number of instances of a given component ("processes") can run asynchronously with each other.



Example of multiplexing

When computers usually had a single processor, this was useful when a lot of I/O was going on; now that machines usually have multiple processors, this is starting to become useful when processes are CPU-intensive as well. The diagram in this section shows a single "Load Balancer" process distributing data between 3 processes, labeled S1, S2 and S3, respectively, which are instances of a single component, which in turn feed into a single process on a "first-come, first served" basis.

## Simple interactive network



Schematic of general interactive application

In this general schematic, requests (transactions) coming from users enter the diagram at the upper left, and responses are returned at the lower left. The "back ends" (on the right side) communicate with systems at other sites, e.g. using CORBA, MQSeries, etc. The cross-connections represent requests that do not need to go to the back ends, or requests that have to cycle through the network more than once before being returned to the user.

As different requests may use different back-ends, and may require differing amounts of time for the back-ends (if used) to process them, provision must be made to relate returned data to the appropriate requesting transactions, e.g. hash tables or caches.

The above diagram is schematic in the sense that the final application may contain many more processes: processes may be inserted between other processes to manage caches, display connection traffic, monitor throughput, etc. Also the blocks in the diagram may represent "subnets" - small networks with one or more open connections.

## Comparison with other paradigms and methodologies

### Jackson Structured Programming (JSP) and Jackson System Development (JSD)

This methodology assumes that a program must be structured as a single procedural hierarchy of subroutines. Its starting point is to describe the application as a set of "main lines", based on the input and output data structures. One of these "main lines" is then chosen to drive the whole program, and the others are required to be "inverted" to turn them into subroutines (hence the name "Jackson inversion"). This sometimes results in what is called a "clash", requiring the program to be split into multiple programs or coroutines. When using FBP, this inversion process is not required, as every FBP component can be considered a separate "main line".

FBP and JSP share the concept of treating a program (or some components) as a parser of an input stream.

In Jackson's later work, Jackson System Development (JSD), the ideas were developed further.[14][15]

In JSD the design is maintained as a network design until the final implementation stage. The model is then transformed into a set of sequential processes to the number of available processors. Jackson discusses the possibility of directly executing the network model that exists prior to this step, in section 1.3 of his book (italics added):

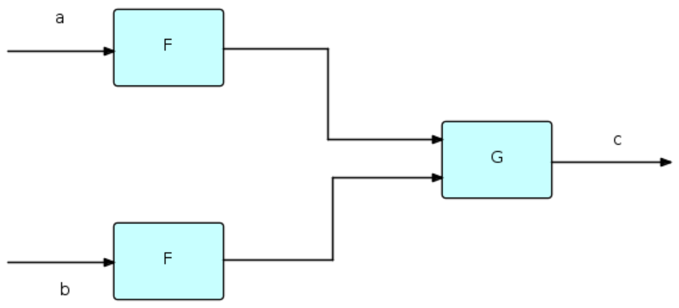
> The specification produced at the end of the System Timing step is, in principle, capable of direct execution. The necessary environment would contain a processor for each process, a device equivalent to an unbounded buffer for each data stream, and some input and output devices where the system is connected to the real world. *Such an environment could, of course, be provided by suitable software running on a sufficiently powerful machine. Sometimes, such direct execution of the specification will be possible, and may even be a reasonable choice*.[15]

FBP was recognized by M A Jackson as an approach that follows his method of "Program decomposition into sequential processes communicating by a coroutine-like mechanism"[16]

## Applicative programming

W.B. Ackerman defines an applicative language as one which does all of its processing by means of operators applied to values.[17] The earliest known applicative language was LISP.

An FBP component can be regarded as a function transforming its input stream(s) into its output stream(s). These functions are then combined to make more complex transformations, as shown here:



Two functions feeding one

If we label streams, as shown, with lower case letters, then the above diagram can be represented succinctly as follows:

```
c = G(F(a),F(b));
```

Just as in functional notation F can be used twice because it only works with values, and therefore has no side effects, in FBP two instances of a given component may be running concurrently with each other, and therefore FBP components must not have side-effects either. Functional notation could clearly be used to represent at least a part of an FBP network.

The question then arises whether FBP components can themselves be expressed using functional notation. W.H. Burge showed how stream expressions can be developed using a recursive, applicative style of programming, but this work was in terms of (streams of) atomic values.[18] In FBP, it is necessary to be able to describe and process structured data chunks (FBP IPs).

Furthermore, most applicative systems assume that all the data is available in memory at the same time, whereas FBP applications need to be able to process long-running streams of data while still using finite resources. Friedman and Wise suggested a way to do this by adding the concept of "lazy cons" to Burge's work. This removed the requirement that both of the arguments of "cons" be available at the same instant of time. "Lazy cons" does not actually build a stream until both of its arguments are realized - before that it simply records a "promise" to do this. This allows a stream to be dynamically realized from the front, but with an unrealized back end. The end of the stream stays unrealized until the very end of the process, while the beginning is an ever-lengthening sequence of items.

## Linda

Many of the concepts in FBP seem to have been discovered independently in different systems over the years. Linda, mentioned above, is one such. The difference between the two techniques is illustrated by the Linda "school of piranhas" load balancing technique - in FBP, this requires an extra "load balancer" component which routes requests to the component in a list which has the smallest number of IPs waiting to be processed. Clearly FBP and Linda are closely related, and one could easily be used to simulate the other.

## Object-oriented programming

An object in OOP can be described as a semi-autonomous unit comprising both information and behaviour. Objects communicate by means of "method calls", which are essentially subroutine calls, done indirectly via the class to which the receiving object belongs. The object's internal data can only be accessed by means of method calls, so this is a form of information hiding or "encapsulation". Encapsulation, however, predates OOP - David Parnas wrote one of the seminal articles on it in the early 70s[19] - and is a basic concept in computing. Encapsulation is the very essence of an FBP component, which may be thought of as a black box, performing some conversion of its input data into its output data. In FBP, part of the specification of a component is the data formats and stream structures that it can accept, and those it will generate. This constitutes a form of design by contract. In addition, the data in an IP can only be accessed directly by the currently owning process. Encapsulation can also be implemented at the network level, by having outer processes protect inner ones.

A paper by C. Ellis and S. Gibbs distinguishes between active objects and passive objects.[20] Passive objects comprise information and behaviour, as stated above, but they cannot determine the *timing* of this behaviour. Active objects on the other hand can do this. In their article Ellis and Gibbs state that active objects have much more potential for the development of maintainable systems than do passive objects. An FBP application can be viewed as a combination of these two types of object, where FBP processes would correspond to active objects, while IPs would correspond to passive objects.

## See also

- Active objects
- Actor model
- Communicating Sequential Processes (CSP)
- Concurrent computing
- Dataflow
- Data flow diagram
- Dataflow programming
- FBD - Function Block Diagrams (a programming language in the IEC 61131 standard)
- Functional reactive programming
- Linda (coordination language)
- MapReduce
- Pipeline programming
- Wayne Stevens
- XProc
- Yahoo Pipes

## References

1. http://jpaulmorrison.com/fbp/introduction.html
2. N. Carriero and D. Gelernter, *Linda in Context*, Communications of the ACM, Vol. 32, No. 4, April 1989
3. N. Carriero and D. Gelernter, *Coordination Languages and their Significance*, Communications of the ACM, Vol. 35, No. 2, February 1992
4. Gabe Stein (August 2013). "How an Arcane Coding Method From 1970s Banking Software Could Save the Sanity of Web Developers Everywhere" (http://www.fastcompany.com/3016289/how-an-arcane-coding-method-from-1970s-banking-software-could-save-the-sanity-of-web-develop). Retrieved 24 January 2016.
5. M.E. Conway, *Design of a separable transition-diagram compiler*, Communications of the ACM, Vol. 6, No. 7, July 1963
6. J. Paul Morrison, *Data Responsive Modular, Interleaved Task Programming System,* IBM Technical Disclosure Bulletin, Vol. 13, No. 8, 2425-2426, January 1971
7. J. Paul Morrison, *Data Stream Linkage Mechanism,* IBM Systems Journal Vol. 17, No. 4, 1978 (http://domino.research.ibm.com/tchjr/journalindex.nsf/e90fc5d047e64ebf85256bc80066919c/2e3ae508088ef61585256bfa00685aff?OpenDocument)
8. W.P. Stevens, *How Data Flow can Improve Application Development Productivity,* IBM System Journal, Vol. 21, No. 2, 1982 (http://domino.research.ibm.com/tchjr/journalindex.nsf/e90fc5d047e64ebf85256bc80066919c/8b4f51d29b014b3685256bfa00685b53?OpenDocument)
9. W.P. Stevens, *Using Data Flow for Application Development*, Byte, June 1985
10. W.P. Stevens, *Software Design - Concepts and Methods*, Practical Software Engineering Series, Ed. Allen Macro, Prentice Hall, 1990, ISBN 0-13-820242-7
11. Johnston, Wesley M.; Hanna, J. R. Paul; Millar, Richard J. (2004). "Advances in dataflow programming

languages" (http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.99.7265). *ACM Computing Surveys*. **6** (1): 1–34.
12. D.P. Friedman and D.S. Wise, *CONS should not evaluate its arguments,* Automata, Languages and Programming, Edinburgh University Press, Edinburgh, 1976
13. http://cecs.wright.edu/people/faculty/pmateti/Courses/7140/Lectures/TelegramProblem/telegram-problem.html
14. "Programming (http://mcs.open.ac.uk/mj665/cernpgmg.pdf)" by M. A. Jackson, published in *Proceedings of Workshop on Software in High-Energy Physics, pages 1-12*, CERN, Geneva, 4–6 October 1982
15. "A System development method (http://www.ferg.org/papers/jackson--a_system_development_method.pdf) Archived (https://web.archive.org/web/20120206005615/http://www.ferg.org/papers/jackson--a_system_development_method.pdf) 2012-02-06 at the Wayback Machine." by M. A. Jackson, published in *Tools and notions for program construction: An advanced course*, Cambridge University Press, 1982
16. "JSP In Perspective (http://mcs.open.ac.uk/mj665/JSPPers1.pdf)" Michael Jackson; JSP in Perspective; in Software Pioneers: Contributions to Software Engineering; Manfred Broy, Ernst Denert eds; Springer, 2002
17. W.B. Ackerman, *Data Flow Languages*, Proceedings National Computer Conference, pp. 1087-1095, 1979
18. W.H. Burge, *Recursive Programming Techniques*, Addison-Wesley, Reading, MA, 1975
19. D. Parnas, *On the criteria to be used in decomposing systems into modules*, Communications of the ACM, Vol. 5, No. 12, Dec. 1972, pp. 1053-8
20. C. Ellis and S. Gibbs, *Active Objects: Realities and Possibilities*, in *Object-Oriented Concepts, Databases, and Applications*, eds. W. Kim and F.H. Lochovsky, ACM Press, Addison-Wesley, 1989

## External links

- Razdow, Allen (December 1997). "Building Enterprise Data Refineries" (http://www.dmreview.com/article_sub.cfm?articleId=689). *DMReview*. Retrieved 2006-07-15.
- Mayer, Anthony; McGough, Stephen; Gulamali, Murtaza; Young, Laurie; Stanton, Jim; Newhouse, Steven; Darlington, John (2002). "Meaning and Behaviour in Grid Oriented Components" (https://web.archive.org/web/20120204225254/http://www.lesc.ic.ac.uk/iceni/pdf/Grid2002.pdf) (PDF). London e-Science Centre, Imperial College of Science, Technology and Medicine. Archived from the original (http://www.lesc.ic.ac.uk/iceni/pdf/Grid2002.pdf) (PDF) on 2012-02-04.
- Black, Andrew P.; Huang, Jie; Koster, Rainer; Walpole, Jonathan; Pu, Calton (2002). "Infopipes: An abstraction for multimedia streaming" (http://web.cecs.pdx.edu/~black/publications/Mms062%203rd%20try.pdf) (PDF). Springer-Verlag, Multimedia Systems. doi:10.1007/s00530-002-0062-3 (https://doi.org/10.1007%2Fs00530-002-0062-3). Retrieved 2006-08-10.
- Kra, David (October 2004). "zSeries and iSeries servers in the grid domain" (http://www-128.ibm.com/developerworks/grid/library/gr-ziseries/). *IBM developerWorks*. Retrieved 2006-07-13.
- Ludäscher, Bertram; Altintas, Ilkay; Berkley, Chad; et al. (September 2004). "Scientific Workflow Management and the Kepler System" (http://users.sdsc.edu/~ludaesch//Paper/kepler-swf.pdf) (PDF). San Diego Supercomputer Center. Retrieved 2006-07-14.
- Bickle, Jerry; Richardson, Kevin; Smith, Jeff (2005). "OMG Software Radio Specification Overview for Robotics" (https://web.archive.org/web/20060714093003/http://www.omg.org/docs/robotics/05-01-06.pdf) (PDF). Object Management Group - Software-Based Communications. Archived from the original (http://www.omg.org/docs/robotics/05-01-06.pdf) (PDF) on 2006-07-14. Retrieved 2006-07-15.
- Blažević, Mario (2006). "Streaming Component Combinators" (http://www.idealliance.org/papers/extrem

e/Proceedings/html/2006/Blazevic01/EML2006Blazevic01.html). *Proceedings of Extreme Markup Languages*. Retrieved 2006-11-09.

- Kauler, Barry (1999). *Flow Design for Embedded Systems, 2nd Edition*. R&D Books/Miller Freeman. ISBN 0-87930-555-X.
- US patent 5204965 (https://worldwide.espacenet.com/textdoc?DB=EPODOC&IDX=US5204965), Guthery, Scott B.; Barth, Paul S. & Barstow, David R., "Data processing system using stream stores", issued 1993-04-20, assigned to Schlumberger Technology Corporation
- Morrison, J. Paul (March 2013). "Flow-Based Programming" (http://ersaconf.org/ersa-adn/Paul-Morrison.php). *Application Developers' News* (1). Retrieved 2014-05-25.
- Staplin, George Peter (2006). "Tcl Flow-Based Programming - TFP" (http://www.tcl.tk/community/tcl2006/abstracts/wed-pm1-4.html). Retrieved 2010-10-07.
- Johnston, Wesley M.; Hanna, J. R. Paul; Millar, Richard J. (March 2004). "Advances in dataflow programming languages" (http://portal.acm.org/citation.cfm?id=1013208.1013209). **36** (1). ACM Computing Surveys. Retrieved 2006-12-05.
- Koster, Rainer; Black, Andrew P.; Huang, Jie; Walpole, Jonathan; Pu, Calton (April 2003). "Thread transparency in information flow middleware" (http://portal.acm.org/citation.cfm?id=777886&dl=acm&coll=&CFID=15151515&CFTOKEN=6184618). **33** (4). ACM Digital Library: Software--Practice & Experience. Retrieved 2006-12-05.
- Stevenson, Tony (February 1995). "Review of "Flow-Based Programming" " (http://www.melbpc.org.au/pcupdate/9502/9502article7.htm). PC Update, the magazine of Melbourne PC User Group, Australia. Retrieved 2006-12-06.
- Lea, Doug (May 2001). "Composing Oneway Messages" (http://g.oswego.edu/dl/cpj/s4.2.html). Retrieved 2006-12-06.
- Bowers, Shawn; Ludäscher, B.; Ngu, A.H.H.; Critchlow, T. "Enabling Scientific Workflow Reuse through Structured Composition of Dataflow and Control-Flow" (http://daks.ucdavis.edu/~ludaesch/289F-SQ06/handouts/7-templates-frames-sciflow.pdf) (PDF). SciFlow '06. Retrieved 2006-12-06.
- Sorber, Jacob; Kostadinov, Alexander; Garber, Matthew; Brennan, Matthew; Corner, Mark D.; Berger, Emery D. (2007). "Eon: a language and runtime system for perpetual systems" (http://portal.acm.org/citation.cfm?id=1322279). Proceedings of the 5th international conference on Embedded networked sensor systems - Session: Power management. Retrieved 2009-01-28.
- Fiedler, Lars; Dasey, Timothy (2014). "Systems and Methods for Composable Analytics" (http://www.ntis.gov/search/product.aspx?ABBR=ADA603097). National Technical Information Service. Retrieved 2014-04-01.