Dataflow Programming
Concept, Languages and Applications

Tiago Boldt Sousa[1,2]
tiago.boldt@fe.up.pt

[1] INESC TEC (formerly INESC Porto)
[2] Faculty of Engineering, University of Porto
Campus da FEUP Rua Dr. Roberto Frias, 378 4200 - 465 Porto, Portugal

Abstract. Dataflow Programming (DFP) has been a research topic of Software Engineering since the '70s. The paradigm models computer programs as a direct graph, promoting the application of dataflow diagram principles to computation, opposing the more linear and classical Von Neumann model. DFP is the core to most visual programming languages, which claim to be able to provide end-user programming: with it's visual interface, it allows non-technical users to extend or create applications without programming knowledges. Also, DFP is capable of achieving parallelization of computation without introducing development complexity, resulting in an increased performance of applications built with it when using multi-core computers. This survey describes how visual programming languages built on top of DFP can be used for end-user programming and how easy it is to achieve concurrency by applying the paradigm, without any development overhead. DFP's open problems are discussed and some guidelines for adopting the paradigm are provided.

Keywords: dataflow programming, visual programming, end-user programming, programming languages, parallel computing

1 Introduction

Dataflow programming (DFP) introduces a new programming paradigm that internally represents applications as a directed graph, similarly to a dataflow diagram. Applications are represented as a set of nodes (also called blocks) with input and/or output ports in them. These nodes can either be sources, sinks or processing blocks to the information flowing in the system. Nodes are connected by directed edges that define the flow of information between them. Most visual programming languages that use a block-based architecture for representing their workflow are indeed based on DFP [3]. Several advantages are inherited with such model, as presented in this paper.

[3] Although UML may seem an obvious candidate, it should not be regarded as a programming language, but rather as a specification language. Methods for making UML executable exist [20], although they are mainly ad-hoc solutions [10] and not part of the core standard, hence, not making UML a visual programming language.

1.1 Motivation

DFP is a commonly forgotten paradigm, despite its ability to successfully solve certain scenarios, from which the author highlights two.

A first advantage is the existence of visual programming languages[4], easing the work of programmers in a tool that, due to its simplified interface, can provide rapid prototyping and implementation of certain systems. Visual programming languages are also known to ease the process of providing end-user programming, where the user of an application is able to modify the behavior of the application in some way. Many languages exist providing such capabilities, as described in section 3. Visual programming has been successfully adopted both by experienced programmers and non-technical computer users (while still experienced), who are able to use those language as a tool to either extend an existing application or to build one from scratch.

A second point in favor of DFP is the implicit achievement of concurrency [16]. In the internal representation of an application, each node is an independent processing block, producing no side-effects, that is, working independently form any others. Such execution model allows nodes to execute as soon as data arrives to them, without the possibility of creating deadlocks, as there are no data dependencies in the whole system. This is a core feature of the dataflow model, removing the need to have programmers handle concurrency issues such as semaphores or manually spawning and managing threads. Such feature can greatly increase the performance of an application when executed on a multi-core CPU, a common architecture nowadays, without introducing any additional work for the programmer.

These two key points from DFP let the author believe that this paradigm should be part of the knowledge of any developer, empowering him to use it in scenarios were it best fits. This survey paper is expected to introduce readers with DFP, describing its historical background, introducing existing languages and open problems, guiding the reader in the right direction to adopt the paradigm.

1.2 Structure

This survey is composed by five sections, from which this first one is the introduction. The history and concepts of Dataflow Programming are described in the next section. Section 3 gives examples of DFP languages, frameworks for implementing the dataflow paradigm and know usages from it. Section 4 argues about some well-known issues over DFP, as well as describing some common answers for some of those questions. In section 5 the author argues on why DFP is relevant knowledge for any developer. Section 6 details future work and the paper is then finished with a last section detailing the conclusions gathered in this survey paper.

[4] Most visual programming languages are based on DFP [25]

2 Dataflow Programming Overview

Dataflow Programming is a programming paradigm whose execution model can be represented by a directed graph, representing the flow of data between nodes, similarly to a dataflow diagram. Considering this comparison, each node is an executable block that has data inputs, performs transformations over it and then forwards it to the next block. A dataflow application is then a composition of processing blocks, with one or more initial source blocks and one or more ending blocks, linked by a directed edge.

2.1 History

DFP has been subject of study in the area of Software Engineering for more than 40 years, with its origins being traced back at at the Ph.D. thesis of Bert Sutherland [30]. Sutherland used a light-pen and a TX-2 computer to create a visual programming language, on top of the SKETCHPAD framework. He also contributed with patterns for graphical representation of procedures that are still used in visual languages today.

In figure 1 Sutherland shows how arithmetic instructions can be represented in both textual and visual forms. In that example, extracted from Sutherland's thesis, we can understand how parallel operations occur and why they result in a reduction of the computation time in even such a small code snippet. We can observe that the calculation of the value of W can be processed simultaneously with the other arithmetic operations occurring in the two vertically aligned nodes, as there are no data dependencies between them. In a DFP language, such parallel computation is achieved automatically by the compiler. The compiler analyses the source and creates an internal dataflow representation of it, based on connected notes, commonly, with each node being processed by an individual thread. DFP compilers exist to create such binaries from either textual and visual languages.

2.2 Architecture

With the increased need to compute large datasets and enable common computers to process more than a single thread at the same time, both in the industrial and scientific world, the need for multi-core processor systems arose [9]. Despite that, multi-threaded programming was still an error prone task to achieve, as it was subject to race conditions, very complex scenarios to debug. The disadvantages and common problems with using threads were well summarized by Ousterhout [24]. Dataflow programming was able to provide parallelism without the increased complexity involved in the management of threads.

In dataflow programming, computation nodes are connected between themselves whenever a node as a dependency on the value processed from another node. Values are propagated as soon as they are processed to the dependent nodes, triggering the computation on them.

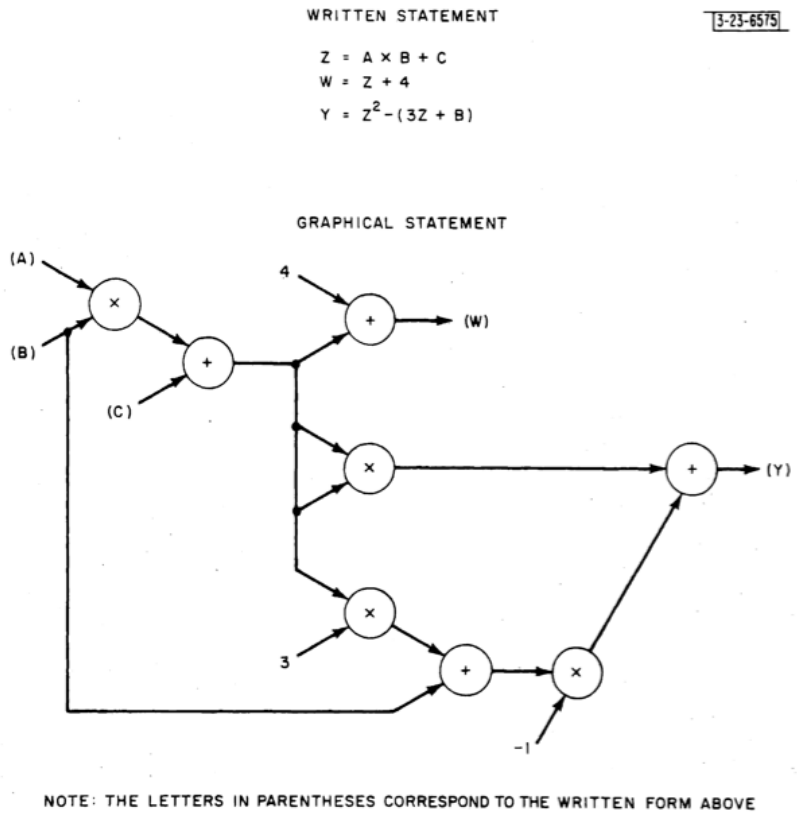

Fig. 1. A comparison of the textual and graphical representation of an arithmetic calculation, from Sutherland's Ph.D. thesis [30].

An initial approach to dataflow programming, by Dennis [6], started by suggesting the use of an architecture able to execute these applications at the hardware level, by giving static memory positions to each node to fill with values that could be read by the remaining nodes that were connected to it. With the introduction of multi-core CPUs and processing farms, languages evolved into supporting this more common architectures for portability reasons and provided developers with the necessary tools to parallelize their computations on common computers [2].

Introduced by Gilles Kahn, the *Kahn Process Networks* approached this problem by having sequential processes (nodes) to communicate via unbounded FIFO queues as message passing protocol [17]. Whenever the entry FIFO queue of a node was not empty, the first value would be processed by the node and outputted into the FIFO belonging to the next node in the chain.

DFP has evolved into a resourceful method to exploit modern computer architectures, composed by multi-core CPUs, as well as computation farms, while reducing the development complexity.

3 Languages and Usages

The dataflow paradigm has been used in a wide range of contexts, supporting either massive computation of data or being the basis for visual languages providing end-user programming capabilities. The *Journal of Visual Languages and Computing* [5] is a reference point in the novel researches being held in this topic.

This section introduces DFP languages and relevant implementations using them. The section describes a textual and a visual dataflow language, particularly, SISAL and Quartz Composer. Although, many more exist, with some relevant names such as LabVIEW [31], VHDL [29] or LUSTRE [12].

3.1 Visual and Textual Dataflow Languages

Independently of the representation style adopted by a the language, it is up to its compiler to analyze the provided source and generate an internal dataflow representation that will define how information will flow between nodes. Several architectures for generating the internal model were researched by Johnston et al [16].

Despite this common comparison to dataflow diagrams, as previously stated, DFP is not a synonym of visual programming, although most visual programming languages are based on the dataflow paradigm. In fact, many early dataflow languages had no graphical representation.

The applications achievable with textual and visual languages do not differ, although, choosing the best language for each situation is a key factor to achieve success. Visual programming languages favor the simplicity of a visual representation. Visual programming can also be used to provide an end-user programming interface. Textual languages require more knowledge but are usually faster to work with, as well as provide a more scalable organization of the source code [7].

SISAL SISAL, acronym for Streams and Iteration in a Single Assignment Language, is a derivative of the Val language and it is a text-based functional and dataflow programming language from the late 80's, introduced by Feo and Cann [19,8,9]. The language is strongly-typed, with a Pascal-like syntax for minimizing the learning curve and enhancing readability.

The language intended to compete in performance with Fortran while using the dataflow model to introduce parallel computation in the first multi-core

[5] Available online at *http://www.journals.elsevier.com/journal-of-visual-languages-and-computing*.

machines. It still provided a micro-tasking environment that supported the dataflow architecture on traditional single-core machines.

In order to increase its performance, SISAL's compiler was able distribute computation between nodes in an optimized way. The management of the internal dataflow was fully automatic — the compiler was responsible to create both the nodes and connections between them. In runtime, each node was executed by an independent thread that was always either running or waiting for data to arrive to the node. Data was processed upon arrival and the result forward along the dataflow chain.

In some benchmarks, SISAL was able to outperform Fortran in computation performance [5].

Quartz Composer Part of XCode, the development environment suite from Apple, Quartz Composer is a node-based visual programming language. The language was developed for quick development of applications for processing and rendering graphical data by non-technical users, as it doesn't require programming knowledges [15].

Quartz Composer stands out from other dataflow languages due to its superior graphical editor, as seen in figure 2. The editor provides an intuitive way for users to add, configure and connect nodes in their dataflow. Each node can be either a source, sink or transformation of data and the editor manages type casting automatically.

The language has a very extensive library of components that interacts with the operative system out of the box. Transformation blocks can be connected between any two blocks of information to provide computation over the flowing data.

The editor allow users to create modules without having to write a single line of code and allows these modules to be integrated with applications developed in Cocoa with the XCode suite. It always allows the creation of animations that can either be used as screen savers or played with Quicktime.

3.2 End-User Programming

DFP is behind most Visual Programming languages based on dataflow diagrams. Such languages not only target experienced developers but also non-technical users, providing them with a simplified interface for building applications. In fact, end-user programming is a common usage for dataflow applications, both by using visual dataflow-based editors, such as Apple's Quartz Composer (previously referred) or with spreadsheets, also a form of end-user programming, empowered by the DFP paradigm.

Graph-based Empowered by intuitive interfaces, such as the one provided by Quartz Composer, users are able to extend or create applications without the need to know how to program. This approach usually relies on the use of a set of pre-defined blocks that can be used to compose the diagram, connected by directed edges.

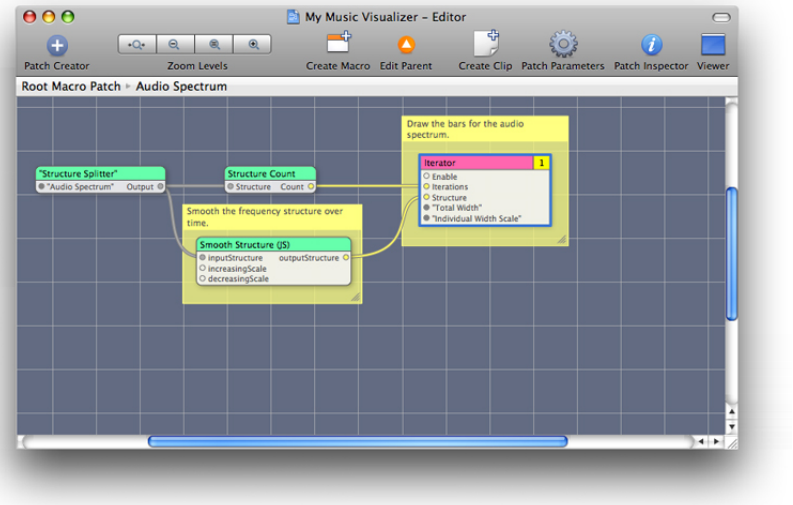

Fig. 2. The Quartz composer editor. Blocks and the connection between them are clearly visible. The interface is visually attractive and easy to use.

Spreadsheets Spreadsheets are probably the most common example of DFP and widely adopted by every type of computer users.

On a spreadsheet, each cell represents a node that can either be an expression or a single value. Dependencies can exist to other cells. Following the dataflow model, whenever a cell gets updated, it sends its new value to those who depend on it, that update themselves before also propagating their new values. This specific type of application is commonly denominated as Cell-Oriented DPF or Reactive programming.

At a more advanced level, tools exist that can extract a visual dataflow model from spreadsheets [13]. These are useful for many scenarios, such as debugging complex expressions or simplify the process of migrating a spreadsheet to a new software.

3.3 The Actor Model

The actor model is a very popular concurrency model by Carl Hewitt from MIT introduced in the '70's. With his team, he researched a method that allow developers not only to simplify the process of parallelizing their computations, but also to increase the confidence on the concurrent behavior of their programs [14].Twitter as adopted it for scaling their computations [21].

An *Actor* is an agent that receives and sends messages, behaving independently from other actors in the system. On each message, the actor is able to start new actors, compute data or reply with messages to other existing actors. In the

dataflow paradigm, an actor is the equivalent to the node and the messages past are equivalent to the connections between nodes.

This architecture perfectly fits the dataflow model when an actor is used as a processing node and the massages between them as communication channels. In cases where there's the need to use an imperative or functional programming language, the actor model could be applied to port the concepts of dataflow programming into those languages, as it has been done by [27,18,11,23].

Many implementations of the actor model are freely available for several languages [26,28,32,1].

4 Open Problems

Dataflow programming is an area still open to further research, with some open issues to answer. In fact, most of the open questions today have long been identified and despite the improvements, patterns for answering them are yet to be achieved.

4.1 Visual Representations

Despite DFP being achievable without a visual programming environment, a graphical representation of how nodes connect in a dataflow-based application provides the user with a better understanding of what the application is supposed to do, providing the possibility of end-user programming. Although, representing conditions and iterations, as well as more complex algorithms or applications might result in a graph with an huge number of nodes with tangled connections, hard to read and maintain. Bellow, some solutions for this problem are proposed.

Iteration and Conditions To represent conditions or iterations as a set of nodes can easily result in a complex graph, nontrivial to understand, if the proper abstractions are not adopted.

Mosconi [22], summarized techniques adopted by the languages Show-and-Tell and Labview, while also introducing his approach to iteration and conditions using the VIPERS language, another dataflow visual programming environment based on the Tcl language [3]. In his paper, Mosconi describes viable implementations of the loop expressions *For* and *While* and explains how index-based iterations can be represented, as well as how to handle ending conditions using blocks with that sole purpose. The representation of a *While* block in VIPERS is shown in figure 3. Similarly, he suggests the creation of a single block for each type of loop and condition, native in the language, in order to significantly reduce the size of the graph, removing the large number of elements that would be needed to construct such expressions.

Visual Granularity Another open problem with visual DFP languages also happens with complex applications, when composed by a very large number of

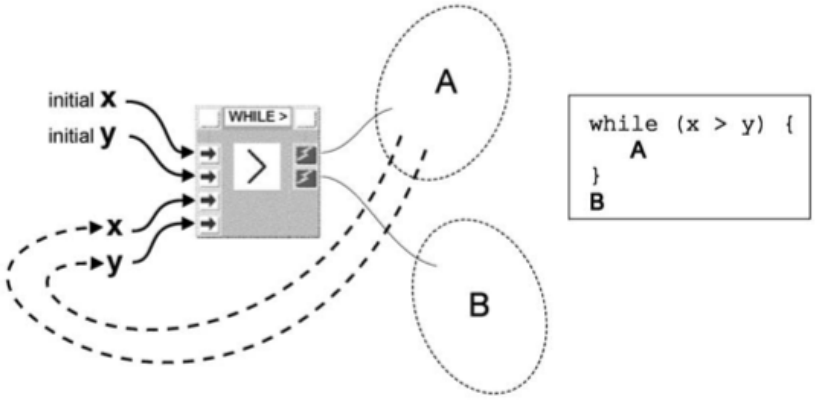

Fig. 3. A While block in VIPERS. A represents a block (or set of blocks) inside the loop that receives and generates new values of x and y. Whenever A returns an $x \leq y$ the loop exits and continues execution to block B.

nodes. In some cases, for experienced programers, the complexity of interpreting a visual representation can end up being higher than reading textual source code.

An approach to solve this problem it to allow a variation on the granularity of data shown at a given moment. To do so, nodes can be grouped hierarchically, so that they can be reduced into a single block that represents them, only showing the inputs and outputs of the whole group of that node. The amount of data shown for a node at a given time can also be configured. At any time the node can be expanded, enabling the user to alter its containing nodes.

4.2 Debugging

Debugging parallel applications requires tools capable of monitoring everything happening in each concurrent operation. In visual programming languages that process becomes even more complex, as the programmer has no direct control over the parallelism. There is the need to map the execution in the direct graph in order to provide visual feedback to the programmer. Browne et al [4] described an approach to debug these languages as a set of five steps:

1. Identify and select the portions of the graph whose behavior will be monitored;
2. Specify the expected execution behavior for each of the nodes in the specified set to be monitored;
3. Run the application with a test scenario as input and capture the execution behavior of the selected portions of the program;
4. Determine where the actual execution and expected events first diverge;
5. Map the elaborated graph of expectations back to the original graph, signaling where errors were detected.

The steps above can be followed by language designers to guide the development of visual debugging tools for DFP languages using a graph-based representation of the application, obtained from either a textual of visual language.

5 Discussion

This paper introduces the DPF paradigm and presents the two most relevant features within it: DFP as a basis for most visual programming languages, including a as way of providing end-user programming in applications and the ability to seamlessly provide developers with a parallel computational model, without introducing development complexity.

Visual Programming Languages allow experienced users to perform rapid application development and non-technical users to extend their application, what is commonly denominated by end-used programming, or create their own applications, without requiring programming knowledges. A common issue with these languages is the complexity to provide abstractions capable of representing an application without resulting in a huge, unperceivable, dataflow diagram — this paper identifies two patterns that can be applied to prevent this situation. Non visual DFP languages also exist. The textual approaches to DFP have a compiler capable of inferring the internal dataflow representation of the application, defining how parallelism is achieved automatically.

Concurrency is also easily achieved by the lack of side-effects in a DFP processing node. Following the concept that data is transmitted as a message and that these are sequentially processed as they arrive to a node provides DFP languages with parallelism out of the box, a valuable feature for developers looking to increase performance on parallelizable applications and algorithms.

6 Future Work

Despite the advantages in performance provided by DFP and the possibility of providing end-user programming with visual languages, there are no frameworks that provide integration of these features in modern day languages. Future work will consist on the development of such framework, believed to be of interest either for academic and industrial purposes, by using the actor model for implementing the dataflow paradigm, independently form the language chosen for implementation.

7 Conclusions

To conclude, the author believes that dataflow programming is a viable paradigm to be explored today for creating either end-user programming and parallel

computation applications. Due to the lack of good quality visual editors and frameworks available for creating such systems, the creation of a generic framework for building end-user programing systems on top of a DFP architecture based on the actor model would be of use in several scenarios and will be pursued as future work.

References

1. Agha, G.: Actors: a Model of Concurrent Computation in Distributed Systems, Series in Artificial Intelligence (Jun 1985)
2. Arvind, D.: IEEE Xplore - Dataflow architectures and multithreading. Annual review of computer science (1986)
3. Bernini, M.: VIPERS (1994)
4. Browne, J., Hyder, S., Dongarra, J.: IEEE Xplore - Visual programming and debugging for parallel computing (1995)
5. Cann, D.: Retire Fortran? A debate rekindled (1991)
6. Dennis, J.B.: Data Flow Supercomputers. Computer 13(11), 48–56 (1980)
7. Erwig, M., Meyer, B.: Heterogeneous visual languages-integrating visual and textual programming pp. 318–325
8. Feo, J., DeBoni, T.: A tutorial introduction to sisal (August 1991), `https://waimingmok.wordpress.com/2009/06/27/how-twitter-is-scaling/`
9. Feo, J., Cann, D.: A report on the Sisal language project (1990)
10. Ferreira, H., Aguiar, A., Faria, J.: Adaptive Object-Modelling: Patterns, Tools and Applications. In: Software Engineering Advances, 2009. ICSEA '09. Fourth International Conference on. pp. 530–535 (2009)
11. Gu, R., Janneck, J., Bhattacharyya, S., Raulet, M., Wipliez, M., Plishker, W.: Exploring the Concurrency of an MPEG RVC Decoder Based on Dataflow Program Analysis. Circuits and Systems for Video Technology, IEEE Transactions on 19(11), 1646–1657 (2009)
12. Halbwachs, N., Caspi, P., Raymond, P., Pilaud, D.: The synchronous data flow programming language LUSTRE. Proceedings of the IEEE 79(9), 1305–1320 (Sep 1991)
13. Hermans, F., Pinzger, M., van Deursen, A.: Breviz: Visualizing Spreadsheets using Dataflow Diagrams. arXiv.org cs.SE (Nov 2011), 9 Pages, 5 Colour Figures; Proc. European Spreadsheet Risks Int. Grp. (EuSpRIG) 2011 ISBN 978-0-9566256-9-4
14. Hewitt, C., Bishop, P.: A universal modular ACTOR formalism for artificial intelligence. 3rd IJCAI-73 (1973)
15. Inc., A.: Quartz composer user guide (July 2007), `http://developer.apple.com/library/mac/#documentation/graphicsimaging/conceptual/QuartzComposerUserGuide/qc_intro/qc_intro.html#//apple_ref/doc/uid/TP40005381`
16. Johnston, W., Hanna, J.: Advances in dataflow programming languages. ACM Computing Surveys (CSUR) (2004)
17. Kahn, G.: The Semantics of a Simple Language for Parallel Programming. In Information Processing ́74: Proceedings of the IFIP Congress (1974), pp. 471-475. pp. 471–475 (1974)
18. Lee, E., Parks, T.: Dataflow process networks. In: Proceedings of the IEEE. pp. 773–801 (1995)
19. McGraw, J.: The VAL Language: Description and Analysis (1982)

20. Mellor, S.J., Balcer, M.B.J.I.: Executable UML: A Foundation for Model-Driven Architectures. Addison-Wesley Longman Publishing Co., Inc. (Jun 2002)
21. Mok, W.: How twitter is scaling (June 2009), https://waimingmok.wordpress.com/2009/06/27/how-twitter-is-scaling/
22. Mosconi, M.: ScienceDirect - Computer Languages : Iteration constructs in data-flow visual programming languages. Computer languages (2000)
23. Oh, H.: Constant Rate Dataflow Model with Intermediate Ports for Efficient Code Synthesis with Top-Down Design and Dynamic Behavior. Quality Electronic Design, 2008. ISQED 2008. 9th International Symposium on pp. 190–193 (2008)
24. Ousterhout, J.: Why threads are a bad idea (for most purposes) (1996)
25. Petre, M.: ScienceDirect - International Journal of Human-Computer Studies : Mental imagery in program design and visual programming. International Journal of Human-Computer Studies (1999)
26. Philipp Haller, F.S.: Actors in Scala pp. 1–139 (Mar 2011)
27. Plishker, W., Sane, N., Bhattacharyya, S.: A generalized scheduling approach for dynamic dataflow applications. In: Design, Automation & Test in Europe Conference & Exhibition, 2009. DATE '09. pp. 111–116 (2009)
28. Scherer, A., Gandhi, R.: Programming Concurrency on the JVM
29. Sjoholm, S., Lindh, L.: VHDL for Designers. Prentice Hall PTR, Upper Saddle River, NJ, USA (1997)
30. Sutherland, W.: On-Line Graphical Specification of Computer Procedures. (1966)
31. Travis, J., Kring, J.: LabVIEW for Everyone: Graphical Programming Made Easy and Fun (3rd Edition) (National Instruments Virtual Instrumentation Series). Prentice Hall PTR, Upper Saddle River, NJ, USA (2006)
32. Vajda, A.: Programming Many-Core Chips - András Vajda, Mats Brorsson, Diarmuid (CON) Corcoran - Google Books (2011)