DISSERTATIONS IN
FORESTRY AND
NATURAL SCIENCES

Maija Marttila-Kontio

Visual Data Flow Programming Languages

Challenges and Opportunities

Publications of the University of Eastern Finland
Dissertations in Forestry and Natural Sciences

UNIVERSITY OF
EASTERN FINLAND

MAIJA MARTTILA-KONTIO

Visual data flow programming languages: challenges and opportunities

Publications of the University of Eastern Finland
Dissertations in Forestry and Natural Sciences
No 30

Academic Dissertation
To be presented by permission of the Faculty of Science and Forestry for public examination in the Auditorium L22 in Snellmania Building at the University of Eastern Finland, Kuopio, on September, 16, 2011,
at 13 o'clock noon.

Department of Computer Science

Kopijyvä
Kuopio, 2011
Editors: Prof. Pertti Pasanen, PhD Sinikka Parkkinen, Prof. Kai-Erik Peiponen

Distribution:
University of Eastern Finland Library / Sales of publications
P.O. Box 107, FI-80101 Joensuu, Finland
tel. +358-50-3058396
http://www.uef.fi/kirjasto

ISBN: 978-952-61-0417-1 (printed)
ISSNL: 1798-5668
ISSN: 1798-5668
ISBN: 978-952-61-0418-8 (pdf)
ISSNL: 1798-5668
ISSN: 1798-5676

Author's address:
University of Eastern Finland
Department of Computer Science
P.O.Box 1627
70211 KUOPIO
FINLAND
email: maija.marttila@uef.fi

Supervisors:
Professor Kaisa Sere
/AAbo Academi University
Department of Information Technology
Joukahaisenkatu 3-5
20520 TURKU
FINLAND
email: kaisa.sere@abo.fi

Ph.D. Mauno Rönkkö
University of Eastern Finland
Department of Environmental Science
P.O.Box 1627
70211 KUOPIO
FINLAND
email: mauno.ronkko@uef.fi

Reviewers:
Professor Joost Kok
Leiden Institute of Advanced Computer Science
Leiden University
P.O.Box 9512
2300 RA Leiden
The Netherlands
email: joost@liacs.nl

Professor Tapio Salakoski
University of Turku
Department of Information Technology
Joukahaisenkatu 3-5
20520 TURKU
FINLAND
email: tapio.salakoski@it.utu.fi

Opponent:
Professor Jyrki Nummenmaa
University of Tampere
Department of Computer Sciences
Kanslerinrinne 1
33014 Tampere
FINLAND

ABSTRACT

This thesis introduces the challenges and opportunities that visual data flow programming language (VDFL) presents to the field of software development.

Some of the commonly recognized advantages of VDFL are comprehensible program code representation and parallel program execution. Despite of these advantages, VDFL has not enjoyed great popularity except for measurement and control system implementations. One of the reasons for this is related to VDFL's restricted applicability for large-scale applications.

In order to enhance VDFL's applicability, it is necessary to first identify the language's restrictions and challenges which may have been hidden behind questionable development methods that can hinder VDFL's advantages, for example, by hiding the parallel program execution. Recent research into the restrictions and challenges of VDFL has remained minimal.

In connection with this issue, this thesis studies the challenges and opportunities that are discoverable in practice. The results of the study have been derived mostly from the implementation of an automated documentation system (ADS), the versions of which were implemented by avoiding the use of "quick-and-dirty" methods that can break the data flow paradigm. The study confirms the existing research results regarding the advantages of VDFL. It also reveals a set of challenges to the use of VDFL, which include the lack of run-time program variability, monolithic program structures, and the lack of advanced design methods. The study also raises the question of end-to-end system verification.

The study suggests new theoretical solutions for discovered challenges; these include: dynamic case structure, event switch, and dynamic computational node. The use of formal methods has been proposed for the system verification.

Universal Decimal Classification: 004.4«236; 004.042
Keywords: visual programming; parallel languages; data flow computing; data flow analysis, flow visualization, data acquisition, health care

Preface

This thesis is the result of the work carried out at the Department of Computer Science, University of Eastern Finland. The work has been funded by University of Eastern Finland (previously University of Kuopio), De Nederlandse Organisatie voor Wetenschappelijk Onderzoek (NWO), and Academy of Finland. I appreciate all the fundings I have received for my research.

I want to thank my supervisors professor Kaisa Sere and doctor Mauno Rönkkö. I especially thank my supervisor professor Matti Nykänen for his intelligent advices and attention he paid to my work. I also want to thank my reviewers professor Joost Kok and professor Tapio Salakoski for their valuable comments.

My colleagues Marko Hassinen, Risto Honkanen, Paavo Pakoma, and Piia Räsänen, who also have become my dear friends, deserve my gratitude. Conversations with you guys have never been boring. Also I warmly thank my other colleagues at Kuopio campus, especially for listening my recent whining about university bureaucracy. Our secretaries (the souls of every well-working department) Merja Leppänen and Leila Tiihonen saved me from many desperate situations. Thank you so much!

At last, I'd like to thank my family: Äiti, Tapio, Tytti, and Olli. Your support has been priceless.

Mikko, I'm grateful for your patience, love, and overall wisdom.

Kuopio May 5, 2011 *Maija Marttila-Kontio*

LIST OF PUBLICATIONS

This thesis is based on the following published articles:

I Hassinen M, Marttila-Kontio M, Saesmaa M, Tervo H. Secure two-way transfer of measurement data. In: Latifi S, ed. Proceedings of the Third International Conference on Information Technology: New Generations, 2006. ITNG 2006, Las Vegas, Nevada, USA, 10-12 April 2006, p. 426-431. Los Alamitos, California: IEEE, 2006.

II Hassinen M, Marttila-Kontio M. Disaster relief coordination using a documentation system for emergency medical services [cd-rom]. In: 1st International Conference on Pervasive Computing Technologies for Healthcare 2006, Innsbruck, Austria, Nov. 29 - Dec. 1, 2006, p. [9 p.]. EMB, 2006.

III Hassinen M, Marttila-Kontio M. Wireless system for patient home monitoring. In: 2007 2nd International Symposium on Wireless Pervasive Computing, San Juan, Puerto Rico, 5-7 February 2007, p. 442-446. IEEE, 2007.

IV Hassinen M, Marttila-Kontio M. A highly automated documentation system: component design. In: Wang PP, Sohn S, Dhillon G, Lee J, Ko FIS, Nguyen NT, Bi J, Sakurai K, Ahmadi M, Ji Na Y, Chung W-Y, eds. Proceedings. Third 2008 International Conference on Convergence and Hybrid Information Technology. ICCT 2008, Busan, South Korea, 11-13 November 2008, p. 382-388. Los Alamitos: IEEE, 2008. Vol. 1.

V Marttila-Kontio M, Hassinen M. Implementation of an automated documentation system with a visual data flow programming language. In: First International Conference on the Applications of Digital Information and Web Technologies. ICADIWT 2008, Ostrava, Czech Republic, August 4-6, 2008, p. 68-73. IEEE, 2008.

VI Marttila-Kontio M, Hassinen M, Kontio M. A monolithic program vs. modifiability: enhancing a visual data flow program with object-oriented techniques. In: Boness K, Fernandes JM, Hall JG, Machado

RJ, Oberhauser R, eds. Proceedings. The Fourth International Conference on Software Engineering Advances. ICSEA 2009, Porto, Portugal, 20-25 September 2009, p. 346-352. Los Alamitos: IEEE Computer Society, 2009.

VII Marttila-Kontio M, Rönkkö M, Toivanen P. Visual data flow languages with action systems. In: Ganzha M, Paprzycki M, eds. Proceedings of the International Multiconference on Computer Science and Information Technology, Mragowo, Poland, October 12-14, 2009, p. 589-594. Los Alamitos, CA: IEEE Computer Society Press, 2009. Vol. 4.

VIII Marttila-Kontio M, Honkanen R. Not-so-free data flow in a visual data flow programming language. In: Li W, Zhou J, eds. Proceedings 2009 2nd IEEE International Conference on Computer Science and Information Technology, Beijing, China, August 8-11, 2009, p. 613-619. IEEE, 2009. Vol. 1.

Throughout the overview, these papers will be referred to by Roman numerals.

AUTHOR'S CONTRIBUTION

Paper **I** was jointly authored with 25% of work on each of four author. The section of measurement application implementation was authored by Maija Marttila-Kontio.

Paper **II** was equally authored (50% + 50%) with specific fields of responsibilities. The documentation system implementation part was authored by Maija Marttila-Kontio, while the security and resource management parts were authored by Marko Hassinen. The application presented in the paper was solely implemented by Marttila-Kontio.

Paper **III** was authored by Maija Marttila-Kontio, jointly with Marko Hassinen. The three sections describing the monitoring system and its implementation was authored by Maija Marttila-Kontio. The security part was authored by Marko Hassinen. The presented application was solely implemented by Maija Marttila-Kontio.

Paper **IV**: The implementation part was authored by Maija Marttila-Kontio and the security part was authored by Marko Hassinen. The rest of the paper was jointly authored.

Paper **V** was 97% by Maija Marttila-Kontio's work.

Paper **VI** was mostly (97%) authored by Maija Marttila-Kontio, who also solely implemented the represented application.

Paper **VII**: With the exception of the section on action system theory, the paper was authored by Maija Marttila-Kontio.

Paper **VIII**: With the exception of the optical torus section, the paper was authored by Maija Marttila-Kontio, who also solely implemented the presented application.

Contents

1 Introduction

The main features of visual data flow programming languages (VDFLs) are visual syntax representation and program execution based on a data flow paradigm. In practice, this equates to a highly concrete and comprehensible language in which visual (graphical) program parts can be executed in parallel.

Visual data flow languages have enjoyed great popularity among various kinds of data acquisition, measurement, and control system implementations [1–3] and have been praised in favor of fast program development through rapid prototyping. Furthermore, the comprehensible syntax representation and predefined programming tools have been found to be usable for end-user programming and novice programming. Despite the advantages, the VDFL has remained largely unknown among the general public.

Currently, at a time when parallel computation has -again- become popular research topic, VDFLs have an advantage over conventional programming languages (PLs) because of their natural support of parallelism [4–7]. It is reasonable, therefore, to make the VDFL more applicable and its purpose more general, and consequently less unknown. Unfortunately, most of the enhancement proposals were produced in previous decades. For instance, VDFLs have been improved with iteration and condition structures borrowed from conventional control flow PLs. A procedural abstraction represents another step towards a more usable VDFL. The previous solutions did not fully comply with the data flow paradigm, but they did considerably increase the usability of the language.

More recent VDFL research, such as [8–13], does not have much to offer in terms of enhancing the VDFLs. Most of the current VDFL research emphasizes cognitive topics, as is the case in [12], while the more technical VDFL research has remained marginal. Furthermore, the technical VDFL literature has focused on the presentation of small or medium-sized measurement applications, as in [14–16]. As a result, it is not possible to reach any conclusions about the VDFL's usability on a larger scale.

This study presents restrictions and problems in VDFL programming,

a topic that has received little attention in the literature. The work is based largely on an empirical work within an *automated documentation system*, ADS. The ADS is a wireless measurement system for patient monitoring. Another application that is important for the study has been implemented in order to visualize the behavior of an optical torus. The restrictions and problems have been discovered by excluding global, local and shared variables when implementing the ADS and the optical torus visualization system. The use of variables has been popular in general VDFL research because of their ability to increase the program's flexibility. In [14, 17–23], for instance, variables have been used for more fluent communication between program parts. In VDFLs, however, variables are against the single assignment rule and they also hide the data flow. Therefore, in order to conserve the VDFL's advantages, such as the natural support of parallelism and program comprehensibility, it is reasonable to exclude the variables.

The prototypes of the ADS and the optical torus visualization application have been implemented with National Instrument's LabVIEW [24]. In a relatively small group of commercial VDFLs, LabVIEW represents one of the most popular and advanced language and programming environments [1, 2, 16, 25–28].

1.1 THE AIMS OF THE STUDY

One of the over-arching purposes of this study is to represent the opportunities and challenges that a VDFL can bring to software engineering, especially to programming. Within this, the three main aims are as follows:

1. To report the advantages and opportunities that VDFL brings to programming. The automated documentation system has been used as a case example.

2. To introduce the challenges and restrictions of VDFL. The main goal is to identify concrete and current examples of the restrictions of VDFLs. This issue, among others, is addressed by excluding the usage of variables in programming. The challenges and restrictions have been discovered during the design and implementation of the automated documentation system prototypes and the optical torus visualization

system.

3. To introduce solutions and suggestions for VDFL's restrictions that are identified and the programming problems that occurred.

1.2 THE ORGANIZATION OF THE THESIS

This study is organized as follows. The chapters 1, 2, and 3 provide the theoretical bases for the following concepts:

- the von Neumann and the data flow execution models in Chapter 2,
- visual programming in Chapter 3, and
- visual data flow programming languages in Chapter 4.

Each of these chapters provides the main characteristics, opportunities and challenges of the case in question.

Chapter 5 presents LabVIEW, a visual data flow programming language, as well as the challenges and opportunities related to the language.

Chapter 6 introduces an empirical work and the results of this study.

Chapter 7 contains the summary of the papers *I*-*VIII*. In each paper, the basic idea and main results are presented.

Finally, Chapter 8 offers conclusions, discussion, and proposals for future research.

2 On execution models

Every programming language is based on a model of a computing system. Two common models, the von Neumann model and the data flow model, are introduced below.

2.1 THE VON NEUMANN MODEL

The von Neumann execution model is defined as follows [4, 29]:

Definition 2.1 *The von Neumann model exploits global addressable memory for storing and modifying program and data objects. During program execution, the memory is updated by program instructions. A single instruction counter (program counter) sustain the address of an instruction to be executed next. Because of the single instruction counter, computing always proceeds sequentially and in a predefined order.*

Conventional programming languages, such as, Java, C and C++ are based on the von Neumann model. The conventional programming languages usually use names, that is, *variables*, to refer a certain locus on memory.

Due to the strictly controlled execution order defined by the programmer, the von Neumann -based languages are also known as *imperative* or *control flow* languages.

2.1.1 Challenges and opportunities in the von Neumann execution model

The main limitations of the von Neumann model are its memory latency, overhead synchronization overhead, and sequential computation [4,29–31]. The single program counter prevents the execution of multiple instructions at the same time. Backus [29] termed this a *bottleneck of the von Neumann model*. Equation 2.1 examines the computation of a simple task as an example of the bottleneck.

$$\frac{x^2 - (x+y)}{(x+y)^2\sqrt{y}} \tag{2.1}$$

A control flow -based pseudo code named *Fragment 1* contains seven instructions:

```
Fragment 1
        [1] k=x^2;
        [2] l=x+y;
        [3] m=square_root(y);
        [4] p=k-l;
        [5] r=l^2;
        [6] s=r*m;
        [7] t=p/s;
```

The variables $k, l, m, p, r, s,$ and t are used to store temporary values into the memory. There are many alternative execution orders for the instructions. For example, instructions 1, 2, and 3, as well as instructions 4, 5, and 6 can be executed in arbitrary order. Whatever order is used, the result of the computation remains the same. In each order, the computation is completed in seven time units. Because the instructions do not depend on each other in instructions 1, 2, 3 and in instructions 4, 5, 6, they *could* be executed at the same time. The von Neumann model however, requires sequential scheduling.

Let us further consider a program that is not related to the previous Fragment 1, but which uses the same variable s:

```
Fragment 2
        ...
        s=s*2;
        ...
```

Considering the previous program fragments, which appear on the same program, the definition of the fragments' relative execution order is essential. Because the variable s refers to the same space in global memory, an

incorrect execution order can cause a semantic error. During the execution of Fragment 1, for example, the result of the Equation 2.1 can be incorrect if the variable s has been simultaneously modified in Fragment 2.

While the usage of variables prevents parallelism, it also simplifies the communication between program parts. By updating the global memory, the updated value is immediately available for other program parts and variables. The usage of variables also enables the program to be flexible and dynamic. Among other things, this represent an opportunity to modify the program behavior during runtime. For example, the content of a *case* structure can be modified using a variable.

Despite Backus' description of the von Neumann model as "complex, bulky and not useful" [29], the execution model sets the basis for most current programming languages. One reason could be related to its sequential and understandable nature [32]. Another is the general purpose problem domain that the universal machine enables. In addition, data abstractions are easy to implement in von Neumann -based languages. According to Ryder et al. [33], abstract data types include encapsulating or enclosing data, naming the data type, placing restrictions on the use of the operators, having rules that specify the visibility of data, and separating the specification from the implementation.

2.2 THE DATA FLOW EXECUTION MODEL

The data flow execution model is defined as follows [4, 34]:

Definition 2.2 *In the data flow execution model, all processing is performed by means of instructions that are applied to values. Instead of the predefined scheduling, an instruction can be executed as soon as it has received all needed input data.*

Although the data flow model does not take a position on the representation of program syntax, data flow execution is usually illustrated with a *data flow graph*. Figure 2.1 is a data flow graph that represents the execution of Equation 2.1.

The execution of Equation 2.1 is completed in four time units and proceeds as follows. After x and y have received the tokens from, for example,

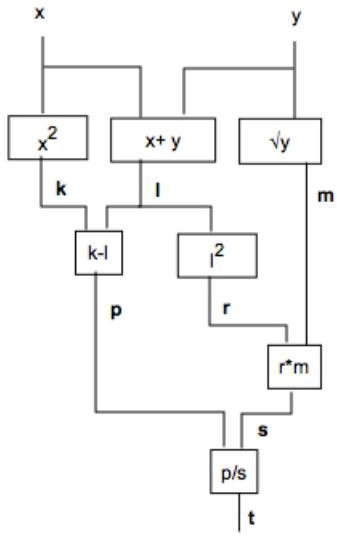

Figure 2.1: A data flow graph computing Equation 2.1.

the environment or another instruction, the tokens flow in parallel to instructions 1, 2, and 3. Because all required data is initially present in each of the three instructions, the instructions are then executed simultaneously. Following the firing rule of data availability, instructions 4 and 5 are executed in the second time unit. Because instruction 6 must wait for the output value from instruction 5, it is executed in the third time unit. Again, instruction 7, is dependent on the value coming from instruction 6, and is therefore executed last, in the fourth time unit. After execution, the result is displayed in t.

Based on Definition 2.2, the main characteristics of the data flow execution model and in data flow languages are as follows [4]:

1. *Data dependencies equivalent to scheduling*.
   The only *sequencing constraint* controlling the execution order is the data dependency between instructions. In other words, the instructions can be executed in parallel if they are not dependent on each other.

2. *Single assignment of variables.*
   In data flow languages, variables can be perceived as *values* that are produced by instructions behaving as *functions* [35]. The single assignment means that once a value is created it can never be modified. Therefore an instruction never contains any functionality for *changing* an input parameter. Each time, the instruction has received all input values, it creates *new* values as outputs. The similar single assignment rule can be found among declarative programming languages, such as functional and logic programming languages [1] [29, 37, 38]. For example, in reference to Fragment 2, above, the statement $s = s * 2$ is absurd in the sense of the data flow because the incoming s can not be modified. Replacing s with another name, such as w, means that the statement is then $w = s * 2$ and it becomes semantically valid.

3. *Locality of effect.*
   The locality of effect means that only short-range data dependencies appear between instructions. The short dependencies are again caused by the absence of global data storage. In contrast to the von Neumann model, the previous problem of the same variable usage in the two unrelated fragments does not appear in the data flow model. If the output value s is used elsewhere in the program, the data arc represented by s is then simply branched into each instruction where the value is needed. Figure 2.2 illustrates a flow graph of the merged fragments.

4. *Freedom from side effects.*
   The behavior of an instruction in the data flow model is functional. Because nothing can be modified, the instruction does not create any side effects elsewhere in the program and the value is guaranteed to remain the same everywhere it is used. As Figure 2.2 illustrates, a "variable" representing the data arc in question, cannot represent multiple different values in different parts of the program.

[1] In actual fact, functional languages are a superset of data flow languages [36].

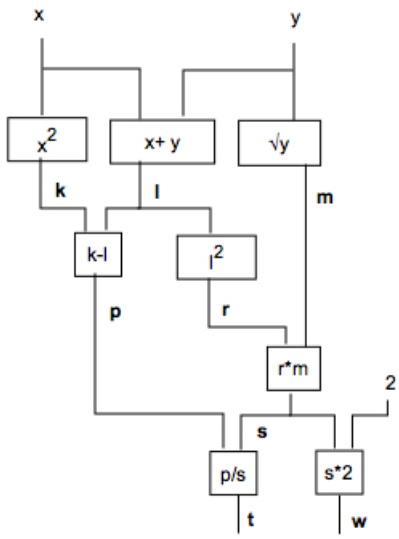

Figure 2.2: Fragments 1 and 2 merging together in a data flow graph.

5. *The lack of history sensitivity in procedures.*
   The lack of history sensitivity means that instructions cannot "remember", because they have no state variables retaining data from one invocation to the next.

2.2.1 Two approaches of the data flow model

The data flow model presented above is based on a *data driven approach* [6, 34, 39]. In such an approach, the execution is dependent on the *availability of data*. In the data driven -based process, an instruction is executed when its inputs have been populated by new data. During its execution, the instruction absorbs data from its inputs and emits new data tokens on its outputs.

A *demand driven approach* enables an instruction to be executed only when needed [39,40]. In this approach, the execution of a program is started at "the end", that is, from the final outputs requesting data from an instruction or instructions to which they are attached. The instruction's input then

requests data from other instructions, etc. In this way, the demand driven approach sets up a network that only contains the necessary instructions.

2.2.2 Challenges and opportunities in the data flow execution model

Because the data flow model was originally designed to exploit "all the potential for concurrency" [41], the implicit parallelism and asynchronism are considered to be the most effective advances of data flow languages [2, 6, 41–43]. Another advantage of the data flow execution model is that it allows program definitions to be represented *graphically* [4, 34]. For example, Figure 2.1 can be seen as a graphical data flow program that computes Equation 2.1. Deviating from the common data flow graph representation, the data arcs in Figure 2.1 are named according to the variables in Fragment 1. Each "variable" represents the output value of an instruction. Although variables are not used in the data flow model, data arcs can be seen as typed variables [2]. A data arc does not have a corresponding memory locus in the conventional sense, but is more like a hardware data bus [7].

Ambler [37,44] has stated that programs with single assignment involve fewer concepts, are easier to understand, have a flow of data that is easier to visualize, and have better-preserved notions of mathematics.

A drawback of the data driven approach is the execution of all executable instructions even though some instructions do not necessarily need to be executed at the specific moment [3]. The data flow paradigm has also been seen as being capable of causing problems, if the problem at hand does not fit into the data flow diagram representation [1].

In practice, the pure data flow model is not usable because it does not support control constructs, such as, iteration or condition structures. The iteration can be considered as contradictory for the single assignment rule [37].

2.3 SUMMARY AND DISCUSSION

As Iannucci [30] stated, the sequential instruction execution of the von Neumann model can be considered antithetical to parallel processing. One-instruction-at-a-time processing has been stated [29] as a bottleneck of the von Neumann model, causing strong latency of memory requests. Also,

sharing data between multiple processes while maintaining functioning synchronization represent a widely known restriction of the von Neumann model. Despite its disadvantages, the majority of today's programming languages are based on the von Neumann model.

The strengths of the data flow model, and the model itself represent a complement to the von Neumann -based execution model. The single assignment rule means that the data flow model is a sound, simple and powerful model for parallel processing [31]. Furthermore, a real advantage of the data flow model is the opportunity to represent the program code graphically. Therefore, the model has been found to be suitable for visual programming languages [4, 34].

The data flow model can be seen as a special case of *Kahn's Process Networks* (KPN) model [40, 45]. The KPN model is an asynchronous, deterministic, and concurrent programming model where processes work autonomously and they communicate over unbounded FIFO data channels. As in the data flow model, the state of an autonomous process in KPN is also inaccessible to other processes and there is no global scheduler present. The KPN model uses *blocking-read* synchronization protocol, which means that a process stalls until the input channel contains sufficient data tokens. Writing to a channel is *non-blocking* and can be done right after the process is executed. Although KPNs are not particularly studied in this thesis, it is worth mentioning that studied restrictions and enhancement proposals in KPNs (see, for example, [46, 47]) are quite similar to those represented in this study.

3 Visual programming

Visual techniques are standards in early phases of software development that exploits various kinds of diagrams (use cases, class hierarchies, state diagrams, entity-relationship diagrams, etc.) [48, 49]. However, these tools have generally been used more for designing, not programming. It is reasonable, therefore, to distinguish the visual language and a visual *programming* language into separate concepts.

A visual programming language (VPL) is often misunderstood as a synonym for any programming language that has a *visual programming environment*. While it does have a visual programming environment, the VPL actually signifies a language that consists of visual *syntax*. VPL is defined as follows [48, 50–52]:

Definition 3.1 *A visual programming language uses pictures to express computation. The program syntax is represented with at least two dimensional program elements. The elements are usually characterized by color, dimension, location, and shape.*

Visual program syntax can be represented in various ways, the most common of which is through *arcs and boxes*. Two less popular methods are *datasheets* (*spreadsheets*) [53, 54] and *programming by demonstration* [55, 56]. This study focuses on the arcs and boxes form of representation.

3.1 VISUAL SYNTAX REPRESENTATION AND BASIC CHARACTERISTICS

According to the arcs and boxes representation, a visual program resembles a *directed graph* that consists of *computational nodes* and *data arcs* between them. The data flow graph illustrated in Figure 2.1 can be also seen as a visual program. Depending on which VPL is in use, computational nodes can also be referred to as *procedures*, *icons*, *box*, *functions*, or *virtual instruments*. *Wire*, *link*, and *line* are synonyms for the data arc. In VPL, text is not needed for naming the nodes, and data arcs also remain

nameless. In order to explain the correspondence of variables in different execution paradigms, the data arcs are exceptionally named in Figure 2.1.

The computational node can be a *phantom node*, an *operational unit*, or a control structure. The phantom node represents an input or output interface between the program and its environment. In Figure 2.1, x, y, and p/s represent phantom nodes, while input and output nodes appear on both the program code and the user interface.

Operational units range from primitive functions to more complex structures that represent elements such as, subprograms. An iteration node and a conditional node represent control structures that typically appear in a visual program.

A computational node can contain zero or more *data ports* (*terminals*). A data arc represents a certain, predefined data type according to the data port to which it is attached. Data values, referred to as *tokens*, flow via the data arc as a discrete *token stream*. In conventional text based programming languages each word can be seen as a token [50]. The token stream is always one-directional flowing from a node's output data port to another node's input data port or to data ports of multiple nodes.

The common characteristics of visual syntax representation are as follows [50]:

1. *Concreteness.*
   Concreteness refers to the use of real values rather than a description of possible ones [57]. The visual program code follows the WYSIWYG principle ("What you see is what you get"). Abstractions cannot be represented visually. Here, a *procedural abstraction* makes an exception.

2. *Directness.*
   Directness relates to a cognitive perspective of programming and it means a short distance between program manipulation and required action. The directness enables direct manipulation of a program [53, 58]. In [58], Shneiderman stated "*the pleasure in using these systems stems from the capacity to manipulate the object of interest directly and to generate multiple alternatives rapidly*".

3. *Expliciteness.*
   Directly stated semantic, whether in textual or visual programs, remove the possibility of misinterpretation. An example of explicitness in visual program code is the way in which relationships between nodes are represented with directed data arcs.

4. *Immediate visual feedback.*
   When a programmer edits the program code, the semantic feedback is automatically and immediately provided by the programming environment. Immediate feedback relates to a program's *liveness*, as introduced by Tanimoto [59].

3.2 CHALLENGES AND OPPORTUNITIES IN THE VISUAL SYNTAX

The advantages of visual languages are comprehensible syntax representation, easy human-computer interaction [60], language independence, direct manipulation techniques [58], fast software implementation through the use of rapid prototyping [61,62], and program debugging properties [34,50,63]. The VPLs also have fewer syntactic restrictions as a way of expressing the program [60]. Visual notations can also provide better organization and can make information explicit [64].

The weak point of visual programming languages is the lack of visual abstraction mechanisms that are essential for designing scalable programs [50, 63]. Consequently, VPLs are often considered inapplicable for large-scale applications [63]. As Burnett et al. stated in [65]: "Making visual programming language suitable for solving large programming problems often seems to require the very complexities VPLs try to remove or simplify. This is called the *scaling-up problem*". Generally speaking, VPLs can suffer from the common characteristic of it: the concreteness of visual syntax. Burnett et al. introduced the scaling-up problem through three major subproblems: (1) representation issues including problems in static representation, screen real estate and documentation, (2) programming language issues, including problems with procedural abstraction, interactive visual data abstraction, type checking, persistence, and efficiency and (3) issues beyond the coding process.

3.3 SUMMARY AND DISCUSSION

The main functioning of VPL can be crystallized to the WYSIWYG principle. “What you see is what you get” basically refers to a comprehensible and concrete program code. What you do not see, however, is something that cannot be implemented, which represents an inability to use abstractions in programming.

The best-known problem of VPLs, the scaling-up problem, has generated studies around its subproblems, mostly those that deal with the representation issues and, more precisely, the efficient use of screen space. Shizuki [66], for example, exploited a popular fisheye view model for better readability of visual program code. Meyer and Masterson [67] critiqued Prograph’s usability and commended the user’s inability to control the level of code details. A similar problem exists in LabVIEW, where the only option is to capsulate parts of the code into a subVI. The *procedural abstraction* method has been the most popular way of avoiding the scaling-up problem.

Representation issues, such as the study of efficient use of screen space and readability of visual program code, have been quite popular. For example, a *fisheye view* method has been represented to improve the comprehensibility and readability of visual program code [68]. Sui et al [69] introduced a VDFL debugger combined together with an improved fisheye view model. Further, Sui et al. [13] presented an automated refactoring tool for improving visual code maintainability, understandability, and reusability.

The most popular research fields generated around the concept of visual syntax representation encompass educational and psychological studies[1] [33]. The concepts of *novice programming* [70–72], programming by example [56, 73–77], *end user programming* [11], and *cognitive processes in programming* [78] represent research categories in which VPLs have been widely exploited. Some of these studies have taken the data flow paradigm into account as a separate factor affecting the research and its results. The focus of some other discussions is only set into the visual syntax,

[1] Although not separately mentioned in the studies, the used visual programming language often follows the data flow paradigm.

as is usually the case in the cognitive research field.

While novice programming is connected to education, end-user programming focuses on software engineering, specifically on the client. As described in [8], end-user programming means "the practice by which end users write computer programs to satisfy a specific need, but programming is not their primary job function". In end-user programming, VDFL is used to reach a better connection and understanding between the programmer and the client [1]. The cornerstone of both the novice programming and end-user programming can be found from the VPL's direct manipulation techniques [58, 79, 80] and the opportunity for immediate visual feedback [71, 80].

Recent psychological research among VPLs has focused mainly on the aspects of human cognition [8, 12, 27, 56, 60, 64, 78, 81]. Human cognition means a study of mental processes, such as problem solving, comprehension, and ways of thinking. The basic questions in this study relate to how different types of programmers understand language representation and how this can be exploited, for example in education and interpersonal communication.

One of the most widely referenced studies on human cognition is Thomas Green's and Marian Petre's [78] "Usability Analysis of Visual Programming Environments". A range of secondary literature [12, 82–86] has been produced to test and develop the cognitive dimensions' framework. In addition, Baroth and Hartsough presented an interesting objection to Green's and Petre's work [78] in [1].

4 Visual data flow programming languages

Integrating the principles and characteristics of visual programming and the data flow execution paradigm, reaches the following definitions (adapting [3, 4, 37, 52]):

Definition 4.1 *A visual data flow programming language (VDFL) contains visual, multi-dimensional objects for conveying semantics. Operation of the visual object is functional and the execution of objects is based on the data flow execution paradigm.*

Unlike the pure data flow programming language, the VDFL usually contains a specific notation for iteration.

Definition 4.2 *The specific notation of iteration consists of:*

- *Definitions of the initial values of the loop controls,*
- *A test to determine whether the loop is to terminate or to continue,*
- *Some expressions giving the value or values to be returned, if the loop is to terminate*
- *Some expressions giving the new values to be assigned to the loop controls.*

Based on the definitions above, the data driven VDFL has the following characteristics and function rules [52]:

1. The operation of a node is functional.
2. A node is executed as soon as its inputs are populated by new data.
3. Data flows via data arcs as a stream of discrete *data tokens*.

4. A node can have zero or more input data ports. Respectively, a node can have zero or more output data ports.

5. If the input ports do not exist, the node is executed once as soon as the program executes.

6. An executed node produces new values for all its output data ports.

7. In order to be executed again, a node must receive new input values for each of its input data ports.

8. Each data port is attached to a single data arc.

9. Data arcs cannot fuse together, but a data arc can be branched into multiple data arcs containing a copy of the original data token(s).

4.1 CONTROL STRUCTURES IN VDFLS

Most VDFLs support control structures for better programmability [3, 52, 87,88]. Although control structures do not follow the pure data flow paradigm, they have been adapted to the VDFLs in favor of program simplicity [3, 52, 89]. As Ghittori et al. stated in [90], "pure data-flow model needs to be enriched with some forms of control flow constructs in order to tackle non-trivial applications".

The basic structures typically offered in VDFLs are *while*, *repeat*, and *for* structures for iteration, and *if-then-else*, *switch*, and *case* structures for condition.

4.1.1 Iterative structures

Iterative structure is represented in VDFLs as a *cyclic data flow graph* [52] (see Figure 4.1). Like the basic node execution, the iteration structure is executed according to data availability. The structure produces a new token(s) that is *cycled back* to its input(s). In order to avoid endless iteration, the structure has a mechanism that terminates the loop.

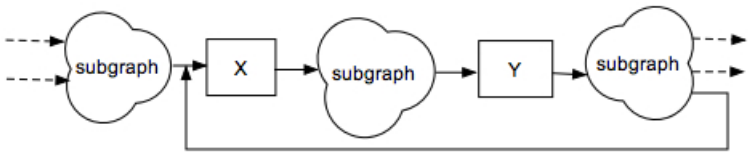

Figure 4.1: General representation of iteration in the VDFL.

4.1.2 Condition structures

Condition structures enable selective routing of data tokens. The simplest "routers" in the VDFL are *merge (selector) gate* and *switch (distributor) gate*. Figure 4.2 illustrates the gates.

The switch gate is a conditional control node that routes data from a single input arc to one of the two output arcs, called the *T* arc and the *F* arc. The incoming control data defines the output arc to which the incoming data is directed. If the control data is true, the data is then directed to the true output arc (the *T* arc). Conversely, if the control data is false, the input data is directed to the false output arc (the *F* arc).

The merge gate operates with three inputs and the input data is associated with a boolean value. If the control data is true, then the data from the true input arc is directed to the output arc. If it is false, the value from the false input arc is directed to the output arc.

Regardless of the strict data flow paradigm, Kosinski [7, 91] and Davis et al. [34] allowed the gates to execute with only a partial number of populated data ports. According to [7,34,91], the switch gate produces an output value only for a single output; similarly, the merge gate can execute after receiving a new token on at least one input data port. While visual data flow languages do not normally have a switch or merge gate some VDFLs do have the same kind of structures for accomplishing the task [3]. For example, LabVIEW offers a select structure that is similar to the merge gate (The select structure and a case structure is presented in Chapter 5.1, Figure 5.3.

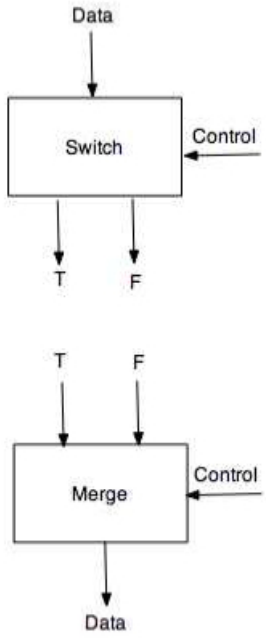

Figure 4.2: The switch and merge gates.

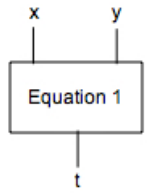

Figure 4.3: Procedural abstraction of the data flow graph represented in Figure 2.1.

4.1.3 Abstractions

The two most important programming language abstractions are *procedural abstraction* and *data abstraction* [50, 65]. The procedural abstraction is essential because it makes the visual program more compact by saving screen space, a frequent problem in visual languages. Hiding numerous program details makes it possible to achieve a higher level of program view [3]. In VDFLs, the procedural abstraction means that any subgraph can be represented as a single function. This is similar to how procedures are used in conventional languages. The subgraph is created by encapsulating the desired part of the program inside boundaries and then attaching a name to it. Data arcs cut by the boundaries represent input and output data ports of the new subgraph. For instance, the function $Equation\ 1$ in Figure 4.3 represents an encapsulated program that has been previously illustrated in Chapter 2.2, Figure 2.1. According to the data flow paradigm, the function $Equation\ 1$ is executed right after the inputs are populated with new data.

The data abstraction means that the programming language supports user-defined data types, data inheritance, and data encapsulation. It is more difficult to exploit data abstraction in VDFLs than it is in conventional programming languages. The reason for this is related to *generality* and *information hiding* which are hard to fit among concreteness and immediate visual feedback [50, 65].

4.2 VDF SOFTWARE ENGINEERING

It can be assumed that the majority of VDF applications have been implemented using the rapid prototyping method. As several general VDFL research articles have mentioned, such as, [28, 36, 50, 61, 64, 92], rapid prototyping with predefined programming tools enables an easy and fast programming process. Having indicated the method's prevalence in many recent publications, rapid prototyping is considered to be not worth mentioning. In this context, many authors [16, 93, 94] seems to feel that the rapid prototyping method goes with the VDFLs *de facto*. Those cases that mention the rapid prototyping method separately (for example, in [94]), praise its low cost and high development speed.

A VDF application structure is generally described with the use of basic

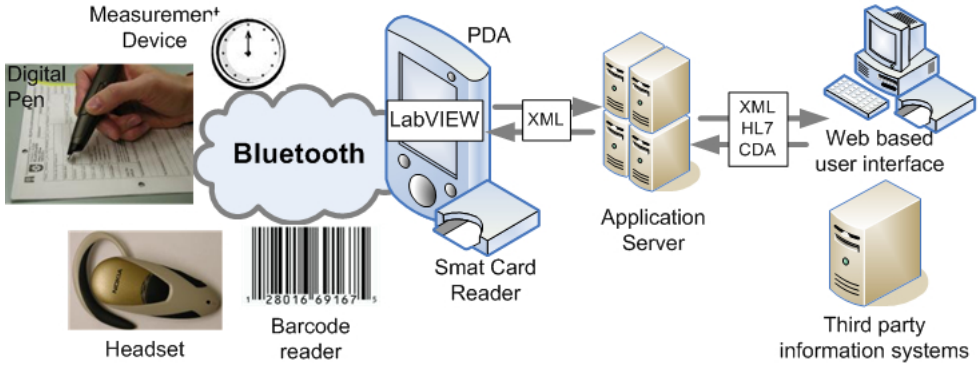

Figure 4.4: A block diagram representing an EMS documentation system.

diagrams. *Flow diagrams* and *block diagrams* represent the most popular design tools. The third diagram worth mentioning is a *state diagram (state transition diagram)*, which is used, for example, in [94]. Hosek et al. introduced the usability of LabVIEW's State Chart Module and Queued State Machine [95]. Examples of the usage of the flow diagram can be found in [14, 96–102].

The block diagram is used to describe both an internal application schema and the hardware architecture, including equipments, devices, network solutions, etc. An example of block diagram is illustrated in Figure 4.4. With the block diagram, authors have described a system at a high-level of abstraction by separating the system structure into basic modules or components, see, for example, [103, 104]. Good examples of how the block diagram has been exploited to illustrate application modules can be found, among others, in [15, 104, 105].

Because of the hierarchical nature of VDFLs [63], a commonly used way of representing an internal application schema is by using a *hierarchical block diagram*. The hierarchical block diagram is generated as a result of *top-down design* method, in which a hierarchical model is reached through an iterative refinement process [7, 33]. The hierarchical structure can also be achieved with a reversed *bottom-up* design method [33]. [106] offers an interesting discussion about the two design methods. The hierarchical structure of VDF programs can help obtain flexibility and better

scalability. As Jackson stated in [18], "hierarchical structure makes it easy to add new components to the model when needed". Examples of the use of hierarchical block diagrams in recent VDF system implementations can be seen in [14, 93, 101].

Over the last decade, more attention has been paid to the design processes of VDF applications. While Simulink and Petri Nets represent common design and simulation tools, different kinds of system architectures and design patterns, such as, producer-consumer, queued state machines, and master-slave patterns are also practical [95, 96, 107–110]. Some general discussions about the role of software engineering in VDFLs include [106, 111]. Also, an interesting comparative evaluation of the usability of Simulink and LabVIEW for design a digital signal processing system is presented in [112].

4.3 CHALLENGES AND OPPORTUNITIES IN THE VDFLS

The symbiosis between the visual data flow programming language and the data flow paradigm produces many advantages. The most obvious advantage of merging the visual syntax together with the data flow execution paradigm is the support of parallelism [31, 34, 113–115]. Another advantage, the fast software development life cycle through rapid prototyping, comes from concrete and comprehensible syntax representation.

As Davis and Keller stated in [34], a directed graph is the most natural way to represent the mental image of data flow execution. For example, data dependencies between instructions are simple and natural to represent with data arcs. Furthermore, the opportunity to insert viewing elements easily at different spots to investigate the data has been found to be usable [3].

Merging program parts into a single program is an advantage derived from the visual syntax representation. Figure 4.5 provides an example of how programs are merged together in the VDFL. Supposing that values x and y are dependent on the outputs of F, the program parts can be merged together by attaching F's outputs to the inputs of the three instructions. Consequently, the phantom nodes x and y can be removed, although all of the necessary information is still present.

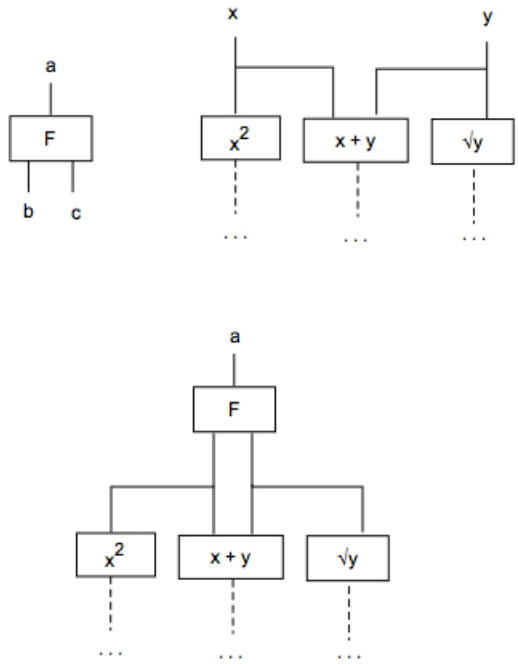

Figure 4.5: Two VDF programs merged into a single program.

For example, Hils [3] stated, that the usability of visual data flow programming languages has been shown to be best among narrow and specialized application domains. Data manipulation, along with signal and image processing, represents typical application domains. The VDFL has also showed its strength among novice programmers or programmers without any programming experience [1, 2].

The main open topics in VDFL research refer to problems in program scalability and general programmability. To the author's knowledge, not many recent studies discuss VDFL's programmability, other than the cognitive point of view. There has been little discussion of any challenges, restrictions, opportunities, possible improvements, and future development trends. Recent studies have only mentioned the VDFL as an implementation tool.

As presented in this study, VDFL's problems and challenges can appear in the form of a monolithic program structure, which is caused by the data flow paradigm. Furthermore, the lack of program dynamics, the absence of a flexible switch structure, and weak run-time program variability are some problems that can cause challenges in VDFL programming.

Good usability of the rapid prototyping method has led to less scrutiny of the VDF software design. Compared to design tools, such as architectures and design patterns used in software engineering in conventional PLs, the scale of the design and modeling tools used in VDF software engineering are more narrow.

The way in which software engineering methods have been used for VDFL poses yet another problem, which has not been studied to any great degree. Most VDF applications are developed for demanding measurement and data acquisition tasks, such as patient care; therefore, a well functioning and robust application must be seriously considered. The fact that most VDF applications have been implemented with rapid prototyping method, creates the need to have a proper testing process.

4.4 SUMMARY AND DISCUSSION

The VDFL has been popular among narrow-scaled application domains, such as testing, simulation, measurement, signal processing, and labora-

tory systems. The language's main advantage has been its easy and fast programming, which does not require advanced programming skills.

Small prototypes of a narrow problem domain do not necessarily expose the challenges and restrictions of the VDFL. Although less reported, programming challenges quickly increase during implementation of larger applications with the VDFL. The general programmability can become clumsy and, at the same time, the program code can turn into monolithic "spaghetti code".

In order to achieve better program scalability and common programmability, VDFLs have been slowly started to enhance with data abstractions. Some VDFLs (such as Data Vis [116]) can provide *higher-order functions* that takes other functions as input values. The more common way has been to use *visual object-oriented programming*[1] VOOP. The idea is based on the possibility of creating user-defined data types (classes) when the corresponding objects can flow as tokens through the VDF program [57]. The concept of *object-flow* and *higher order functions* has been studied in [115, 117, 118].

Unfortunately only a few new publications have actually discussed the usability of object flow in VDFLs[2]. One is [25], which was mentioned above, and another is presented in this study.

In order to guarantee the robustness of VDF application, another solution (instead of a large testing process) is to use formal methods [120]. While formal definition of VDFL have been studied, mostly in previous decades [121–123], publications in VDFL formalization, reasoning and refinement have remained slow. The present study has taken a step on the long path towards automatically verifying VDF applications by presenting a formal modeling with action systems.

[1] Some VDFLs, such as Prograph, Fabrik, and Cantata, supported the object-oriented paradigm.

[2] Several articles have introduced a combination of control flow and data flow languages. The focus of those articles is on the improved code representation and program comprehensibility. See, for example, [119]

5 LabVIEW

LabVIEW (Laboratory Virtual Instrumentation Engineering Workbench) is a commercial software platform and is one of the best-known and widely used visual data flow programming language and software for virtual instrumentation [124]. LabVIEW applications are typically implemented for industrial automation, laboratory research, bio-instrumentation and healthcare purposes [124,125]. The majority of LabVIEW users are scientists and engineers who do not have a strong programming background. Although LabVIEW consists of both a visual data flow programming language and a visual programming environment, the language is commonly considered to be a programming platform.

Referring to the different types of representation of visual syntax, the LabVIEW syntax consists of "arcs and boxes" where *virtual instruments* (VIs) represent the boxes and *data wires* represent the arcs. LabVIEW offers a wide library of predefined virtual instruments, varying from simple algebraic operations to more complex operations for data acquisition, communication management, and database entry, for example. The execution of virtual instruments is based on the data flow paradigm and, more precisely, the data driven approach.

As is the case with most VDFLs, LabVIEW cannot be seen as a *pure* data flow language because of the control structures it offers. Another eye-catching difference can be found between LabVIEW and a pure visual syntax representation. In LabVIEW, the user is able to create textual code and to use local and global variables[1] in programming.

The programming environment of LabVIEW contains a *block diagram* and a *front panel*. The front panel represents the user interface consisting of *controls* and *indicators* (i.e., the phantom nodes). The controls represent virtual instruments (VIs) for input data whereas the indicators are used for displaying output data.

The actual programming is done on the block diagram. In addition to

[1] The use of local and global variables should be avoided. The use can lead unexpected program behaviour and slower performance [126].

the input and output VIs, the block diagram contains *function* VIs. The functions represent predefined instruments in LabVIEW's function library. Each VI has input and/or output terminals (i.e., data ports) for attaching incoming and outgoing data wires.

Figure 5.1 presents a measurement program prototype of the Automated Documentation System (ADS) used in this study. The application reads an oxygen saturation level and pulse information from a wireless pulse oximeter via a Bluetooth connection. The application has six control instruments. Five of these, *Bytes to read*, *Data in*, *Channel*, *Address*, and *Bytes to read 2* are for data acquisition and for managing a Bluetooth connection. With the sixth control instrument, *Stop*, the user eliminates the application execution.

The *Pulse* and *SPO2* indicators are used to display the measurement results. The *Error* indicator displays possible error data. In addition to the basic controls and indicators, the application contains function VIs, such as a function for extracting a subarray from an input array.

5.1 CONTROL STRUCTURES IN LABVIEW

LabVIEW offers *while* and *for* structures for iterative computation management, as shown in Figure 5.2. The continuous data acquisition in Figure 5.1 has been managed with a *while* structure. The data acquisition code to be repeated is placed inside the *while* structure.

A condition is managed in LabVIEW with a *case* structure. For simplified cases, the programmer can use a *select* structure, see Figure 5.3. The *select* structure is based on the merge gate and it selects data from one of the data wires according to the Boolean control value. In Figure 5.1, the *case* structure is used to control the execution of the *while* structure.

5.2 ABSTRACTIONS IN LABVIEW

LabVIEW supports procedural abstraction by enabling the usage of sub virtual instruments (subVIs). The previous application shown in Figure 5.1, used three subVIs to communicate between the system and the measurement device. The subVIs are *Spo config*, *Bluetooth Read*, and *Bluetooth*

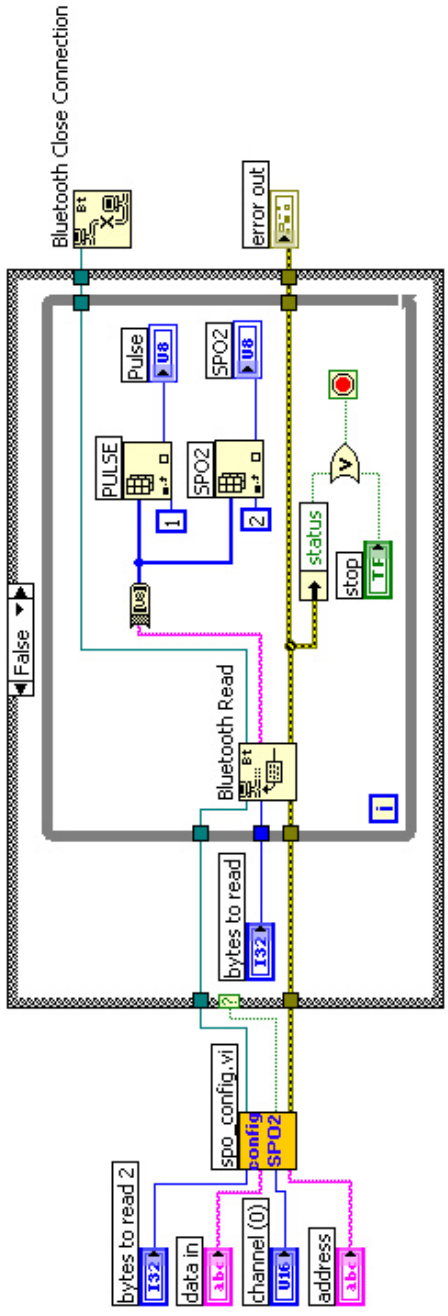

Figure 5.1: A simple example of LabVIEW's block diagram. With this program a user can read his/her pulse and oxygen saturation transmitted wirelessly from a Pulse-SPO2 device.

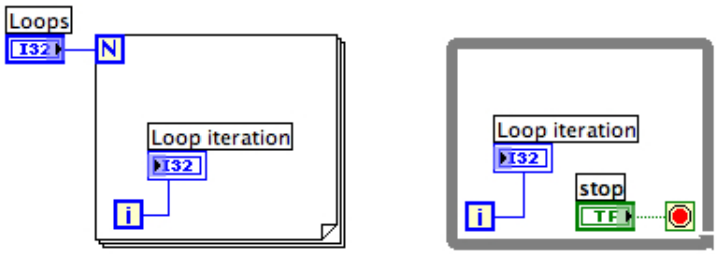

Figure 5.2: LabVIEW's for structure (left) and while structure (right).

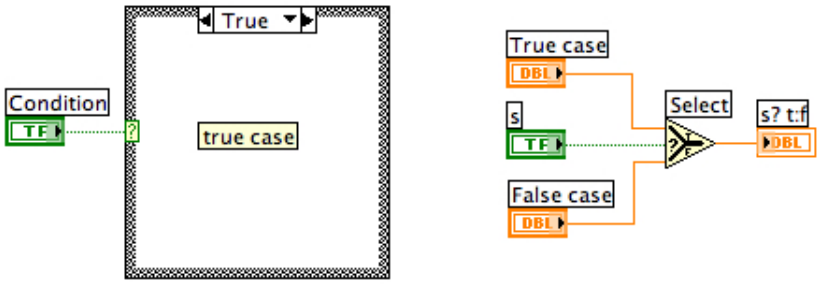

Figure 5.3: Two condition structures in LabVIEW. The CASE structure is on the left and the Select structure is on the right.

Close Connection.

LabVIEW also supports data abstraction. LabVIEW's object-oriented programming tool (LVOOP[2]) allows the creation of user-defined data types and the exploitation of data inheritance and data encapsulation in programming [25]. With the LVOOP, the user can also achieve a cleaner program code that is easier to debug [127]. Applications implemented with the LVOOP tool have been advertised to scale better for large programming tasks [25, 127].

The execution of the application in Figure 5.1 proceeds as follows. The subVI *Spo config* is executed right after it has received data from the four controls. The case structure is executed after receiving data from the three subVIs. If the status of the error is False, the *while* loop is executed next. Otherwise, the Bluetooth connection is closed, occurred errors are displayed, and the application is terminated. Inside the *while* loop, the measurement data is continuously acquired from the pulse oxymeter device until the user terminates the application execution or an error occurs.

5.3 USAGE

LabVIEW has been used for monitoring, data acquisition, data manipulation, controlling, and simulation system development. As is the case with other visual programming languages, LabVIEW has also been used in computer science and engineering education.

5.3.1 Monitoring, controlling, and simulation

LabVIEW applications have been built to measure and/or monitor: soil contamination [103], weather characteristics [96], the transient temperature of droplets [128], air quality [99], properties of cementitious materials [129], and a nodal resistance of composite structures [15]. Chu and Ganz's WISTA-system (wireless telemedicine system for disaster patient care) [130] is somewhat similar to the automated documentation system, which has been a platform for empirical VDFL research represented in this study.

[2]Graphical object-oriented programming GOOP is another commonly used term.

LabVIEW has been used for different kinds of control system implementations. In [131], Bryant et al. presented a system for controlling the measurement of motor and neural data. Hosek et al. [95] introduced a control application for an optical coherence tomography system. Control systems have also been implemented for vehicle navigation [94, 132], controlling the water level in two water tanks [133], and for maintaining heating and air conditioning [134].

Simulation applications represent the third application domain that often appears in LabVIEW application presentations and in the VDFL research. Simulation applications have usually been implemented in collaboration with a simulation specialized system, usually with Simulink of MathWorks. In [135], LabVIEW, Simulink, and Active-HDL were used to control and simulate hardware properties. LabVIEW and Simulink were also used in [14], which introduced a wind power simulation system. In addition to collaborative simulation systems, Gutiérrez-Castrejón et al. presented an interesting comparison between two similar simulation applications implemented with LabVIEW and Matlab [136] . Naturally, VDFL-based simulation systems can be implemented without an external simulation application. In [137], for example, Kim et al. introduced a LabVIEW application for a web-based nuclear reactor simulator.

5.3.2 LabVIEW in education

Naturally, the problem domain of LabVIEW dominates the educational scope [9]. In [138], LabVIEW was used for teaching logic design concepts and practices for computer science and engineering students. In [139], LabVIEW was used for distance and mobile access to remote reconfigurable laboratory workbenches for e-learning within electrical engineering education. Coito [140] again used LabVIEW as a remote laboratory environment for blended learning. Remote access was also used for laboratory experiments in [141] and for experimental tasks on measurement instrumentation in [142]. In [143], LabVIEW and Visual Basic were used to introduce the latest design techniques, computer-aided instrumentation, design, and process control to students. In [26], students used LabVIEW to develop software programs that were able to simulate physical and biological phenomena.

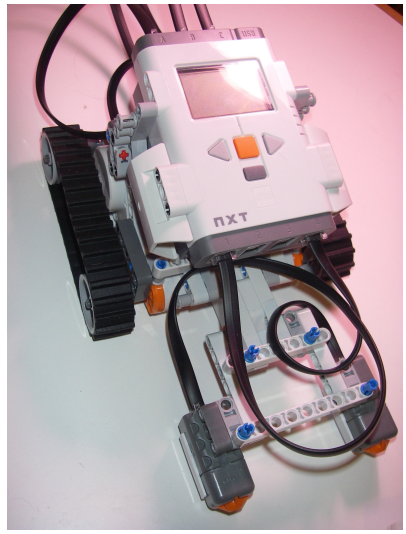

Figure 5.4: The LEGO® Mindstorms® robot car.

From an educational perspective, robotics and VDFLs form a fruitful combination. Teaching robotics with an easily programmable platform and, vice versa, teaching VDFL with the help of programmable robots, have both been sources for many studies. The LEGO® Mindstorms® [144] robot has been a practical tool for teaching computer engineering [145–151], the actual programming of which is managed with a LabVIEW type, drag-and-drop -based programming language. Figure 5.4 illustrates a LEGO robot car. Figure 5.5 presents an example of the program code controlling the car movements.

5.4 CHALLENGES AND OPPORTUNITIES IN LABVIEW

The LabVIEW application developers, such as those listed below, usually reach a similar conclusion: in order to develop a measurement, control, test, simulation, or an analysis application, LabVIEW is a good or even better choice than conventional text-based PLs. The following advantages have been brought out as typical research results:

- A user friendly graphical user interface (GUIs) that is easy to design

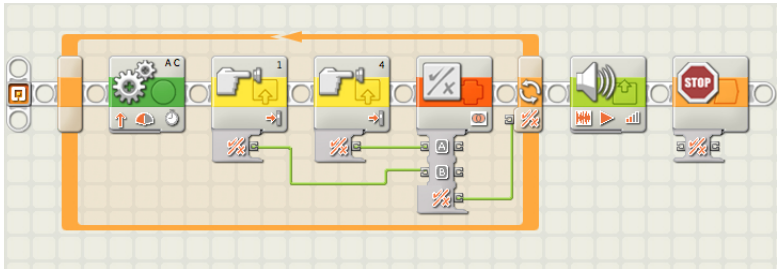

Figure 5.5: A program code of the LEGO® Mindstorms® robot car.

[104, 132]

- System modularity [16, 25, 95, 96, 128, 137]
- Parallel computation [16, 95]
- Easy and fast development process [94, 132, 135, 152]
- Powerful system performance [96, 101, 153]
- Extendability and flexibility [16, 17, 93, 96, 102, 103, 128]
- Low costs [23, 93, 96, 131]

The extendability in the list refers to possibility to modify the program for other kinds of problem domains with little efforts. For example, in [128], Lin and Zhao has mentioned that "transient temperature measurement system can also be used widely in other field of transient temperature measurement's.

Whitley and Blackwell stated in [28] that "LabVIEW users are confident that the advantages of the visual programming provided by LabVIEW outweigh its disadvantages". According to Baroth and Hartsough [1], the main advantages are ease of learning, ease of communication, speed, increased productivity, and adaptability. On the other hand, Baroth and Hartsough questions the language's applicability for large-scale applications which is also known as the scaling up problem [63, 65]. Furthermore, Baroth and Hartsough are skeptical if the implemented applications can be properly maintained over ten years.

The important question has long been whether LabVIEW and other VDFLs are applicable to large-scale applications. A study by Jamal and Wenzel [63] discussed the possibilities of developing complex graphical programs in LabVIEW using hierarchical programming methodologies. They stated that "without effective abstraction mechanism, visual programs fail to scale adequately when compared to popular text-based programming languages". As a result, however, Jamal and Wenzel found LabVIEW applicable to a "broad range of applications" providing "high degree of freedom developing such applications". The recent LabVIEW implementations introduced above all seem to be fit into the group of small and medium-sized applications.

As presented in Chapter 4.3, one of the problems in visual programming was the user's inability to efficiently control the level of details in the program code. A similar problem exists in LabVIEW, for which the only solution is to encapsulate parts of the code into a subVI. However, as presented in this study, encapsulating a program to subprograms can restrict the communication between program parts (and subVIs). Many LabVIEW programmers avoid possible restrictions and problems either knowingly or unknowingly, by using methods borrowed from control-flow based textual languages. Typically, this means using of global and local variables or shared variables. The usage of variables, however, fights heavily against the data flow paradigm. In [154], the usage of global variables is "considered poor programming practice: they hide the data flow of your application and create more overhead". The use of global and local variables in LabVIEW, and more generally in VDF programming, can hide the restrictions and problems of a VDFL would not otherwise be fixed in a more suitable way.

Although LabVIEW supports local, global, and shared variables (see Figure 5.6), programmers are advised to avoid them [126]. In a significant number of recent publications that present a LabVIEW application, the programmers have used global, local and/or shared variables. For instance, global variables have been used in [17–20], local variables in [17, 21–23] and shared variables in [14]. Based on a quite restricted view of the authors' program codes, it is hard to say whether the program could have been developed in a more proper way, or if the use of global and/or local variables

Figure 5.6: From left: local variable, global variable, and formula node in LabVIEW.

was necessary for a functioning VDF program.

Another doubtful way to develop a LabVIEW program has been to use of a particular node that enables parts of the program code to be written with a textual PL. In LabVIEW, the node is called a *formula node* (as in Figure 5.6). Usually the formula node is exploited in situations where LabVIEW does not offer proper tools for implementation. Another case is to use formula node for saving screen space. Examples of the unnecessary use of the formula node can be found in [15, 19]. As is the case in Putta's et. al's application, the usage of the formula node is irrelevant and the textual program code is implementable with virtual instruments like the rest of the program. An example of how Putta et. al [15] have used the formula node is presented in Figure 5.7. Below the formula node in Figure 5.7 is the alternative program code implemented with virtual instruments.

The use of variables and embedded textual program code is usually an attempt to avoid, among other things, the scaling-up problem and sometimes uneasy programming. It also indicates the inadequacy of program structures of the VDFL. It almost seems that problems and restrictions no longer exist because of the abnormal but commonly accepted programming tools and methods in use. However, if variables and embedded textual code are excluded, restrictions and problems will appear in VDFL programming.

5.5 SUMMARY AND DISCUSSION

Several visual data flow programming languages have been brought to the market since 1970 [2, 13, 36], some of the most best-known of which are

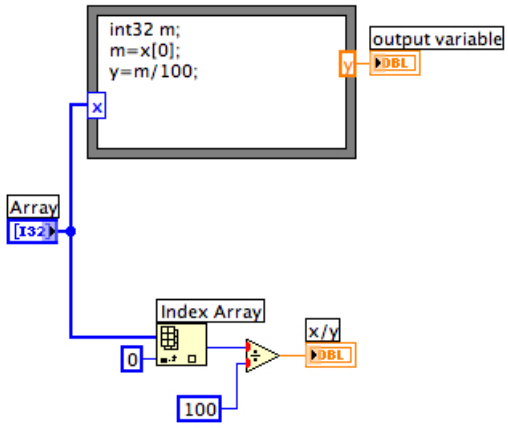

Figure 5.7: Above: Formula node used by Putta et. al. Below: the same calculation implemented with virtual instruments.

listed below. [3] contains an inclusive introduction to various VDFLs. Note that the listed languages that are no longer available are marked with an asterisk (*).

- Fabrik (*) [155, 156] for constructing user interfaces.
- HI-VISUAL (*) [157] for image processing and for office work, such as accounting and data management tasks. The language uses the arcs and boxes representation. Most of the publications relating to the language are from the beginning of the millenium.
- VIVA (*) [59] for image processing. The language uses the arcs and boxes representation.
- Cantata (*) [158] for image and signal processing. The language uses the arcs and boxes representation.
- Vipers (*) [88] for general purpose programming.
- DataVis (*) [116] for scientific visualization. The language uses the arcs and boxes representation.

- Prograph (*) [159] for general-purpose programming. The language uses the arcs and boxes representation.

- VEE [160, 161] for test, measurement, and data analysis. The language uses the arcs and boxes representation. Commercial programming language from Agilent Technologies.

- LabVIEW [24, 162, 163] for test, measurement, and data analysis. The language uses the arcs and boxes representation. Commercial programming language from National Instruments.

In addition to the above list, several recent VDFLs have been developed for academic purposes, such as LabScene [69], NimoToons [164], and PdaGraph [165]. There are also programming languages that are advertised as having a pure visual syntax but where some of the program code has to be *written* despite it being created with visual objects. Examples of such languages are Microsoft Visual Programming Language, MVPL [166], two-dimensional C++ [167], and Simulink [156, 168]. Although Simulink is not a particular focus of this study, it is worth noting that Simulink often appears among VDFLs as a model-based design tool [153]. In the author's opinion, Simulink and MVPL can be seen as such languages, because, in both languages textual code is sometimes required inside a visual node.[3] Also popular are studies that discuss visual languages as *design models* that support the data flow paradigm. Relational Blocks [169] and Ptolemy [170,171] are examples of such languages. Naturally, many visual data flow *programming* languages have been introduced.

While discussing of VDFLs, commonly known *workflow languages* are worth mentioning. Workflow languages (WFLs) are process control languages that were originally developed from the notion of factory automation [172]. Usually workflow languages are modeled and analyzed with graph-based formalism like Petri nets [173, 174] or with Kahn's Process Networks [46]. Workflow languages are used in *workflow management systems* that are used for controlling, monitoring, optimising, and support-

[3] Simulink and Microsoft's MVPL are usually described as VDFLs. The issue of the pure VDFL is relative: depending on how tight the definition of VDFL is, they may or may not be read among VDFLs.

ing, among other things, business processes. *BPEL* and *Taverna* represent examples of such workflow management systems. Commonly known BPEL (Web Services Business Process Execution Language for Web Services) is an executable workflow language. It describes business process activities as Web services [175]. Taverna, again, is used for bioinformatic workflows [176, 177]. Workflow languages have some similarities with VDFLs. A workflow is represented with a directed graph where nodes represent workflow activities and links represent interaction between activities. Like in VDFLs, in workflow graphs, nodes can be seen as functions. Similarly, a workflow can be structured as a hierarchical graph that contains sub-workflow graphs. The execution of a workflow is based on either *control dependency* or *data dependency*. According to the execution rule, workflows are called *control driven workflows* (control flows) or *data driven workflows* (data flows) [178]. The relation of VDFL and the latter execution rule is apparent.

6 Empirical work and the results of the study

In this study, the advantages and disadvantages of visual data flow programming language have been empirically examined through two applications. Both applications have been developed with LabVIEW. During implementation, atypical methods such as, the use of global and local variables have been avoided in programming. Hence, possible restrictions in VDFL are more easy to expose.

The first and larger work discusses a multistage design and implementation process of automated documentation system (ADS). Discovered characteristic of the VDFL have been published in Papers **I**-**VII**. The Section 6.1 introduces the basic structure, functioning, and requirements of the ADS. Discovered advantages and restrictions are also listed in Section 6.1.

The second and smaller empirical work discusses visualization of an optical torus with a visual data flow programming language. The basis here is in exploiting visual program code and the flow of data tokens for demonstrating how data packets flow in an optical torus. It was soon discovered that VDFL's apparently similar model of data flow can not be put into practice for visualizing the torus. The met challenges and restrictions are presented in Paper **VIII** and Section 6.2.

In both studies, solutions for improving the VDFL and enhance its programmability have been proposed. These solutions as well as the discovered restrictions are novel and related research have not been published. Without an access to LabVIEW's source code, the proposed solutions (excluding a *dummy packets* solution) can only be presented in theory. This means that the applicability of the solutions could not have been put into practice.

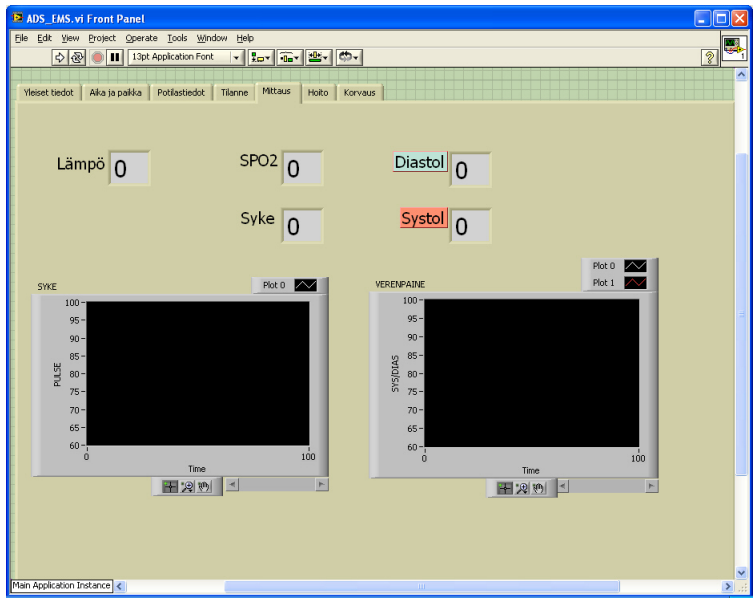

Figure 6.1: The user interface of ADS.

6.1 AUTOMATED DOCUMENTATION SYSTEM

The automated documentation system (ADS) is a system for patient monitoring. The architecture of the ADS is based on secure two-way transfer of measurement data, the structure that is presented in Paper **I**. The systems deploys wireless measurement devices for acquiring data from patient's vital signs and given treatment. A technical overview of the system and its measurement equipments are given in Paper **II**. The ADS has been originally designed for emergency medical services (EMS), but the usability of the system has also been studied for disaster relief coordination, as presented in Paper **II**, and for patient home monitoring, as presented in Paper **III**. The Figure 6.1 illustrates the user interface of ADS. The user interface is based on an official form of the Social Insurance Institution of Finland.

In emergency situations, time is scarce and treating the patient takes all the attention. Considering the later care in a hospital, documenting the treatment process and the patient's condition is similarly important to the

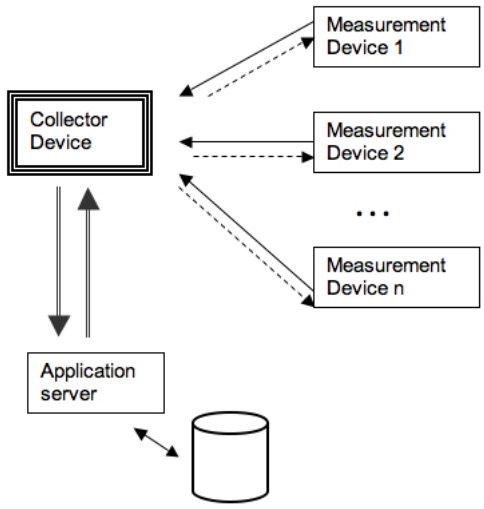

Figure 6.2: The block diagram of the ADS.

treatment process. Still, the documentation is done when the situation allows it. Currently, in Finland, the documentation is mostly done by manually filling an official form (in exception of ECG, for instance). The space for treatment information are highly limited and the data of vital signs is based on few observations taken from time to time. Furthermore, copying the data into hospital's paper form can cause errors and the process is time consuming. For this purpose, the ADS offers more reliable documentation approach. By offering time stamped data about given medication and patient's response to the treatment enables more informative and comprehensive health care research in the field of EMS.

The ADS consists of two separate parts; a collector device and measurement devices (see Figure 6.2). A laptop, PDA or Tablet PC represents a basic collector device that acquires and stores measured data. For good usability, all measurement devices are wirelessly connected to the collector device. In our study, the wireless connection was enabled through Bluetooth communication protocol. Interaction between measurement devices and a collector device is usually one-way but, in some cases, a measure-

ment device can be controlled during runtime. Therefore, in Figure 6.2, the connection from a measurement device to a collector device is presented as a dashed line. The collector device also communicates via secure two-way connection to a server located, for instance, on a hospital or a health care centre.

The field of health care sets high requirements to the automated documentation system. It is important to implement an automated system that does not need extra hands to control it. Furthermore, whenever technology is attached to a person, a system's reliability and robustness are prerequisite.

6.1.1 Early phases on the ADS development

First versions of the ADS system were developed by prototyping. In VDFL based application implementations this is a normal method. As discovered, it is easy to prototype small program parts, such as, data acquisition tools for an oxygen saturation device and a program for controlling a bar code reader. The usability of predefined virtual instruments for easy and fast application development has been brought out in Papers **II** and **III**. Easy modification, for instance, refers to ability of quickly replacing virtual instruments by other ones. However, as the requirements and the size of the ADS increased, it became clear that the rapid prototyping as a single development method does not guarantee a well-functiong result.

In Paper **IV**, the structure of the ADS has been designed by combining methods used in VDF application design with the methods that are commonly used in designing applications based on a conventional PL. Because of the important role that modularity plays in developing modifiable and maintainable systems, the general structure of the ADS has been presented as a four-layer architecture (see Figure 6.3). Layers are from top to down: User Interface, Business Logic, Communication, and Local Storage. On top, a pervasive Security component offers security service to each of the layers. Each layer contains one or more components. In this study, the focus is set on the Communication layer and its components that are: Long Distance Communication (LDC), Short Distance Communication (SDC), and Data Acquisition (DAQ) component.

The general component architecture is language-independent, but the

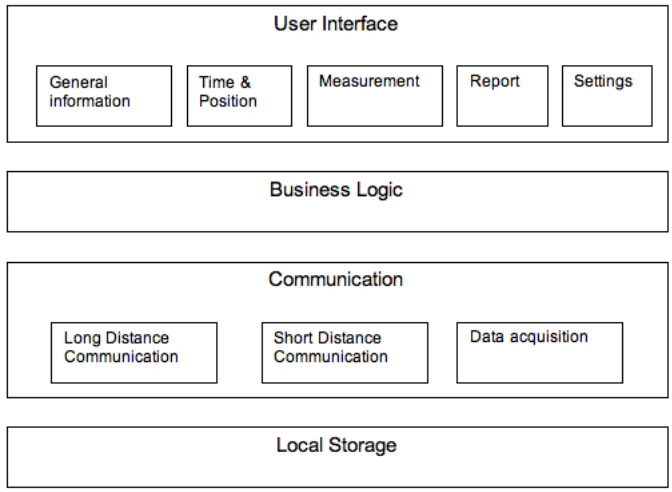

Figure 6.3: The four-layer architecture of the ADS.

internal structure of a component is designed using the top-down design method. The main reason for using the top-down method is that it follows the same hierarchical structure as the VDFL. At that point, at which parts of the ADS had been prototyped, the top-down design method and VI prototyping were found to be usable. This is partly because of the design's ability to implement various alternative prototypes and test programs with minimal effort.

As the process of the ADS development proceeded we learned that using the top-down design method as the only design tool does not necessarily meet all assigned requirements. For example, there is no practical way to ensure whether the designed model can be put into practice. The top-down design method produces a hierarchical structure of program nodes, without accounting for the necessary data types or the communication between program structures.

Compared to the software engineering in the field of conventional PLs, VDFL lacks effective design tools and methods. Despite the VDFL based application design is not similar to conventional design processes, diagrams such as, activity, use case, communication, and sequence diagrams were

used in the design process of the ADS. The sequence diagram and activity diagram were used for understanding the communication between components in different layers. The use case diagram and sequence diagram were used for modeling, for instance, the type of interaction that is needed between the ADS and a health care person. The use case diagram also helped us to notice the actions possible to automatize with the ADS. In addition to the previous diagrams, a component diagram and a class diagram were found usable later in the ADS development process. They were used for creating user defined data types enabled by LabVIEW's visual object-oriented programming (VOOP).

While the conventional diagrams enabled us to analyze and clarify the communication and overall activities, a clear connection between designing and the requirements of the ADS did not reached. Because meant for control-flow based languages, a conventional diagram does not take into account the absence of variables and abstractions in VDFL. For example, the communication between program components is easy to model with the design methods mentioned above, but they do not help to recognize and further avoid component interaction problems caused by the VDFL. An interaction problem between program components are presented in Paper **VI** and it is also explained in Section 6.1.2.

The scarcity of proper design tools and methods has led to the study of an alternative way to verify a robust VDF system. The question is especially important when the ADS is used for patient care. There are no room for mistakes or situations where, for instance, data acquisition does not work.

A study that is presented in Paper **VII** has examined the usage of formal methods, specifically action systems for the verification. The main idea relates to the question of whether the robustness of the ADS can be formally verified. The kind of study is a novel approach that has not been previously examined.

Action systems are based on predicate transformer systems. An action is any statement in an extended version of Dijkstra's guarded command language. The functioning of an action is almost similar to a computational node; while a computational node can be executed in parallel, actions can be executed in any order or in parallel.

The study of action systems and VDFLs is just beginning. After basic mappings have been examined the results are promising. The mappings from the visual computational node, node composition, case structure and merge structure into to action systems have been discovered to be natural because of the similarity in the semantics of VDFL and action systems. The most important similarity is recognized in parallel program execution, although it is important to take differences in execution into account. An executable action will not necessarily be executed at all in an action system, while in the VDFL an executable node is always executed. Another difference can be found in the naming of variables representing data arcs. However, these differences do not prevent or complicate the usage of actions systems in formalizing VDFLs. The results heavily encourage the continuation of work on reasoning and refining VDFL-based programs.

6.1.2 Implementation process

VDFL's restrictions started to expose on a larger scale when prototyped ADS components needed to be connected together. The lack of program's run-time variability and overall weak system dynamics, as well as a monolithic program structure were major restrictions that impeded programming, system's well-functioning, and system maintenance.

The lack of program's run-time variability represents an inability to easily modify program behavior during runtime. A reason for this relates to the omission of global, local, or shared variables in programming. For example, user interface components can be created dynamically during the execution of a program based on conventional programming language. In contrast, all VDF components on the user interface must be created before the program execution and only the *visibility* of a component can be controlled during runtime.

Another example of the lack of variability, a rigid case structure, is introduced in Paper **V**. If again conventional PLs and VDFLs are compared, the content of a case structure in conventional programming languages can be retrieved from a database but a similar procedure is impossible in VDFLs. For example, the case structure in Figure 6.4 is controlled by a *Measurement devices* menu list. Depending on a value picked from the menu list, the case structure executes a corresponding case. The menu list is mod-

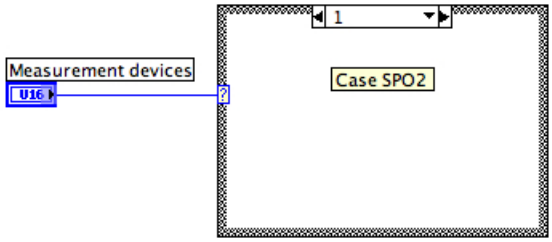

Figure 6.4: A case structure controlled by a menu list control.

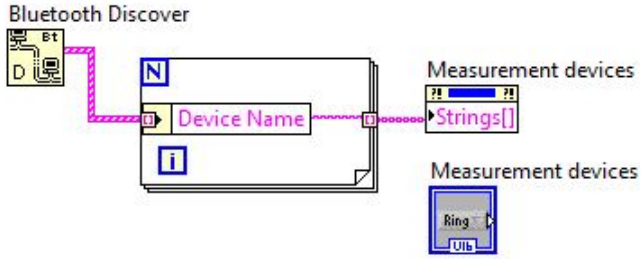

Figure 6.5: A menu list modified by a property node.

ifiable during runtime, that is, it can update its values with the help of *property node*. In this example, the *Measurement devices* menu list is updated according to detected measurement devices in range. Figure 6.5 represents a program code for modifying the menu list with its property node. The property node (on top right) gets its values from the *Bluetooth Discover* node returning the names and addresses of discovered measurement devices in range. The names of devices are collected to an array representing input data of the menu list's property node.

The problem occurs, when the value picked from the menu list is forwarded to the case structure. Unlike the menu list, the case structure can not be updated during runtime because it does not have its own property node. All its cases has to be created before program execution. Therefore, if, for instance, the ADS end user needs to add a new measurement device into

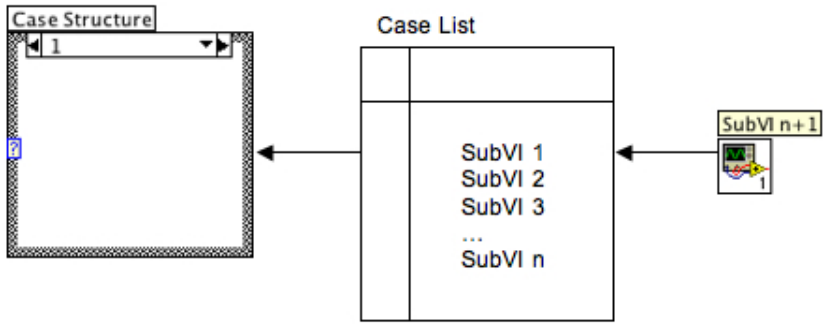

Figure 6.6: In the middle: a list maintaining information about measurement subVIs.

the system, he or she must manually modify the case structure as well. It is obvious that this will not guarantee easy modifiability and maintainability of the ADS.

In order to deal with the case structure problem, Paper **V** introduces a solution entitled a "*dynamic case structure*". The dynamic case structure exploits a property node with which to dynamically create the content of the case structure. The functionality of the case structure modification is similar to the menu list modification. The case structure executes the case *n* according to the input value *n*. The value *n* represents, for example, the name of a subVI to be executed. It is important to note that each case in the case structure must have the same type of input and output data. This restricts the usability of the dynamic case structure. The problem can be avoided by using encapsulated data. LabVIEW's support of encapsulated data are discussed in Paper **VI**, and it is also explained later in this section.

The dynamic case structure involves more internal controlling than the basic menu list modification. The case structure must have rules for managing subprograms that are executed in the case structure. One solution is to maintain a list of all possible subVIs (see Figure 6.6). Each subVI contains a control code of a measurement device. If a new device and its control code are attached to the ADS, user only needs to update the SubVI list.

Yet another example of the lack of program dynamics is a *dynamic call problem* that is described in details in Paper **VI**. The problem refers to in-

ability to manage execution of some subVIs while some others remain idle. This problem was examined to fix with the use of *visual object-oriented programming* (VOOP) tools. The usability of the VOOP tools have been tested in the implementation of the ADS's data acquisition (DAQ) component. The VOOP brings basic object-oriented characteristics such as, data inheritance and data encapsulation, to VDFL programming.

The flexibility and modifiability of the DAQ component was enhanced because of the ability to create user defined data types. For example, defining a data type that is based on a measurement device ancestor class made it possible to dynamically introduce and initialize only the necessary measurement objects.

The dynamic call problem is closely related to the problem of monolithic program structure, the restriction of VDFL that is also described in Paper **VI**. A monolithic program structure refers to a large and clumsy application without any modularity. Modularity, again, is the key of today's applications that are easy to modify and maintain. The monolithic program structure is a naive solution to the component interaction problem. The interaction problem results from the data flow paradigm. For instance, sending or receiving data requires a temporary termination of the emitting and the receiving subprograms (components). This can be avoided only by re-factoring the original subprograms onto the main program. This results in a more monolithic program code that is hard to maintain and modify.

At the same time, the problem of monolithic structure represents VDFL's restriction that could be avoided through the use of variables. However, program parts become tightly coupled if local variables are used. Furthermore, the use of variables weakens the program's modifiability and maintainability; this is because if, for instance, a computational node is removed, the programmer must manually remove each variable referring to the node.

Outside the programming forums, the monolithic structure of VDFL based programs is not widely discussed problem. Although in some forums data encapsulation and data inheritance have been described as usable tools against monolithic VDF program structure, in this study the VOOP tools have not been found to be particularly effective for the purpose. Although it is easy to pass data into a subVI with the help user defined data types, this does not solve the interaction problem between program components.

The main reason for this is related to visual program structures that operate as functions.

6.2 OPTICAL TORUS VISUALIZATION SYSTEM

An optical torus visualization system is based on the exploitation of the visual program *code* and the data flow paradigm. The purpose of this project was to create a visual program that is similar to a 4×4 optical torus model. The torus has four processors and twelve routers. The processors and routers are represented as subVIs in LabVIEW. The basics of 4×4 optical torus and the visualization application are given in Paper **VIII**.

The visualization system continues the discussion about the restricted dynamics of a VDF program. The main problem was caused by the absence of a VDF control mechanism that could have routed incoming data tokens to the desired output data wires. In one particular case, the mechanism should have been able to route a data token to one of the two output data wires. The restriction derived from the *partial dependency* of incoming data and it relates to the structure of switch gate illustrated in Figure 4.2 in Section 4.1.2. As mentioned in Section 4.1.2, the switch gate is not supported at the moment in the form presented by Kosinski and Davis et al. in [7, 34, 91].

In Figure 6.7, a basic router is illustrated. The router has two inputs and two outputs. The problem posed by the partial data dependency reveals a limitation of data flow paradigm. In connection with the data flow paradigm, free flow of data is often mentioned. Anyway, here the execution of a router is strongly limited by the data flow paradigm. The problem occurs due to the rules of routing data tokens. If there is token in the input wire *A in*, it is always routed to the output wire *A out*. The token in *A in* is never routed to *B out*. A token in the input wire *B in* can be either routed to *A out* or *B out*. Which way it is routed depends on the address of a target processor but also the contents of the wire *A in*. If there is data token in *A in*, the token in *B in* is always routed to the wire *B out*. This refers to the *partial dependency* of the *B in* on the *A in*.

A router can have 0, 1, *or* 2 data tokens to route. Under the data flow principle this is not possible, and thus the router waits until it receives data

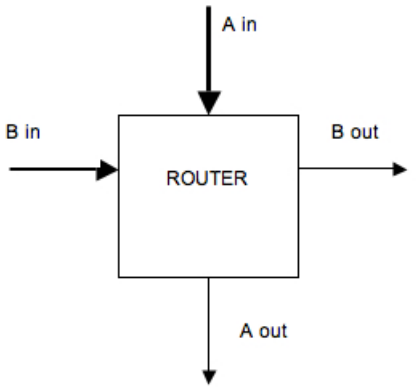

Figure 6.7: A router having two inputs and two outputs.

tokens on both of its input wires. So, the flow of data tokens turns synchronized and not-so-free.

Paper **VIII** introduces three solutions to the problem. Although the research and proposed solution are based on LabVIEW, the results are applicable to other VDFLs, as long as the VDFL supports event handling and the ability to set and get node properties. The problem and the proposed solutions have no related research in publications.

A *dummy packet* is a solution that can be implemented with current LabVIEW. According to the data flow paradigm a computational node is executable only when all its input data wires contain new values. The solution exploits empty values such as, empty strings or empty arrays for enabling the router to execute when only a single input data wire contains an actual data. Here, the actual data refers to a data that is relevant for a router or a processor. The dummy packets do not represent actual data but are rather used to fulfill the data flow. Because lots of empty strings or empty arrays are needed in this solution, the program requires wide resource consumption. Also, the solution does not meet the basic idea of visualizing how data tokens flow in an optical torus because of the empty data. Therefore, the solution of dummy packets is not good.

The *dummy packets* based implementation process raised also the ques-

tion of a *null value*. The null value is common in conventional programming languages and refers to an indeterminate data. In VDFLs the null value does not exists but it would be required to indicate an empty data wire. The previous dummy packets solution becomes more usable if the dummy packets can be replaced by null values. The proposed solutions below also require the VDFL to support the null value.

An *event switch* and a *dynamic computational node* (DCN) represent solutions that currently can not be verified in practice. Both solutions exploit an idea of monitoring the content of a data wire. Furthermore, whenever content monitoring is needed, the use of null value plays an important role. Because data tokens do not arrive into a node's input terminals at exactly the same time, the null value can set the pace to the program's execution. This means a "time window" within which to wait for incoming data. Otherwise, the event switch or DCN, having more than one input data wire, will never execute. The usage of these solutions is not limited to the problem scope presented in the paper. Both solutions can be seen as a modified switch structure that work somewhat like a switch on a train track; in that it can steer the flow of data tokens into the appropriate data wire.

The functioning of event switch is based on advanced event handling and the ability to set and get node properties. An event structure is a basic control structure in LabVIEW that enables an action to execute according to the occurred event. An event can represent a changed value of control or indicator, for instance. A single event structure can manage only one event at a time. However, the proposed event switch solution is required to handle more complicated logical firing rules than the current event structure can handle.

In the event switch solution, four specific nodes *N1*, *N2*, *N3* and *N4* represents data of *A in*, *B in*, *A out*, and *B out*, respectively. Illustrated in Figure 6.8, each of the nodes has a property node placed inside the event switch structure (See Figure 6.9). Diverging from the basic event structure, the event switch can manage more complicated firing rules, such as, "null value in N2 **AND** new value in N1". A corresponding event occurs and the value in the property node N1 is then transmitted to the property node of N4. The event switch now simulates the original router in optical torus that can route 0, 1, *or* 2 data tokens.

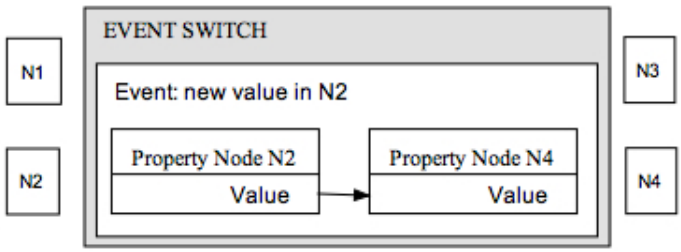

Figure 6.8: The event switch.

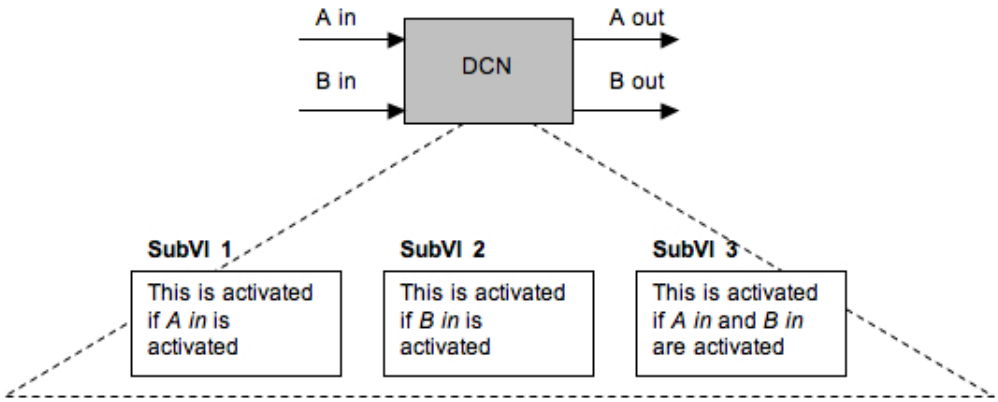

Figure 6.9: The dynamic computational node (DCN).

The ideology of a dynamic computational node (DCN) is quite similar to the dynamic case structure presented in Section 6.1.2 and in Paper **V**. The DCN has dynamic data terminals that can be switched on or off (see Figure 6.9). The way in which the DCN becomes executable depends on the activation rules of a node. This is similar to the execution rules of the event switch structure. The activation rules are individual for each DCN and are defined by a programmer. The DCN differs from a conventional computational node by executing a subVI that is dependent on the activated data terminals. Therefore, if an input terminal *A in* is activated, the DCN executes a corresponding subVI and the token in *A in* is routed to the output terminal *A out*. Now, only the data terminal *A out* is activated and the token flows to another DCN.

It should be noted that wherever the DCN is used, it transitively forces every other node attached to the DCN's data wires to be dynamic. Therefore, a program consist of either DCNs or basic computational nodes.

The advantage of using the DCNs instead of variables can be seen in the visual program code representation. In the DCN, the representation of a program code is more concrete than in the case of using variables. The program that consists of the DCNs can be viewed as a program that is made up of *program "variations"*. The "variation" of the program that is actually executed depends on which data wires are populated by new data tokens. What comes to the data flow paradigm, the solution does not break the paradigm since a program variation is only executable when all its inputs have new values.

6.3 SUMMARY

The common advantages of VDFL were observed, including comprehensibility, ease of use, modularity, and fast implementation process. These advantages have been known for a long time and have been reported in numerous papers, such as, [1, 2, 58, 60–62, 64]. As learned in this study, the implementation of smaller applications or program components is fast, with rapid prototyping and top-down design methods. When implementing larger applications, such as the ADS, however, the use of rapid prototyping and top-down design methods does not guarantee the application's practi-

cality. While the top-down design method offers a good modular structure, it does not offer any guidelines for the practical communication between program components. Despite the emergence of some new design aids, VDFL still lacks effective design tools and methods.

The concrete visual syntax and the data flow paradigm both present challenges to program dynamics. Specifically, the problem derives from the *lack* of program dynamics, which, among other things, can result in an inability to programmatically modify the program or the behavior of a program structure.

For instance, considering a situation where a customer needs to attach new measurement device to a measurement system, the customer is required to modify the system's source code. In many cases, program codes for controlling new devices is offered by the system developer but without variables, the new programs are hard to automatically attach into the system. This can lead to problems, among others, with the *highly automated* documentation system, prerequisites for which are the system's easy modification and maintenance.

The data flow execution paradigm itself can be a restrictive factor that causes a strongly synchronized program execution. Routing data flow in a certain direction requires a more dynamic control structure than what is currently available. One answer could be the switch structure proposed by Dennis in [42]. However, this structure is not offered in modern VDFLs and the output data in this structure can only be directed to one of the two output data arcs. This conflicts with the data flow paradigm, where each output data arc must be populated by new data.

Solutions have been proposed, both in practice and in theory, to deal with the previous challenges in programming. A *dummy packet* represents a solution to the problem of routing data packages in optical torus application. For the problem of program dynamics, *a dynamic case structure*, *a dynamic computational node* (DCN), and an *event switch* have been introduced. The weak point is, the solutions' practical usability and implementability have not been designated. The proposed program structures also follow the data flow paradigm, although in an expanded form.

The runtime modification and the flexibility of visual data flow programs can be partially enhanced by using visual object-oriented program-

ming (VOOP). Features such as inheritance and data encapsulation can significantly reduce the number of data arcs on program code. The true benefit, therefore, comes from the node execution, which now depends only on a single data element or an encapsulated data package. However, the control structures and the concrete representation of a visual data flow program still restrict the interaction between program components. Consequently, the VOOP tool cannot solve the problem of monolithic program structure.

7 Summary of papers

Papers **I-VII** are related to the research of the Automated Documentation System (ADS) and are presented first. Paper **VIII** which is not related to the ADS, is presented last.

Paper I: Secure two-way transfer of measured data

This paper introduces the basic architecture, functionalities, communication solutions, and the implementation tool of the measurement system. The measurement system architecture is developed with three main qualities in mind: secure two-way transfer of measurement data, quick adaptability for different kinds of measurement tasks, and strong data presentation capabilities through XML techniques. The Automated Documentation System (ADS), introduced later, is based on this architecture. Here, the measurement system is implemented with LabVIEW and designed to run on a PDA platform. By exploiting LabVIEW's PDA Module, the prototyping of several measurement system functionalities was easy.

The paper represents the cornerstone of the later research into measurement system implementation with the VDFL. More specifically, the paper sets the basis for the ADS design and implementation processes represented in Papers **II**-**VI**.

Paper II: Disaster relief coordination using a documentation system for emergency medical services.

This paper represented the ADS through more technical details and with the most important requirements. It introduced the measurement devices used, the communication solutions and the basic functionalities for controlling the measurement. The Paper **II** focused on the function of measurement rather than the entire ADS. The measurement process set the basis on a data acquisition (DAQ) component and a short distance component (SDC), which plays an important role in later papers. Furthermore, the system requirements presented in this paper represent the reasons for later challenges

and problems.

Paper III: Wireless system for patient home monitoring

The paper **III** presents an adaption of the ADS's measurement component for patient home monitoring. The contribution of the paper is to represent VDFL's advantages that have been discovered during the implementation of the monitoring system.

The first advantage discovered is the possibility to create personalized user interfaces without a great deal of effort. This advantage comes from the VDFL's comprehensible program code representation and predefined user interface components that are easy to modify or replace by other UI components. The advantage is decisive in situations where the users of the monitoring system (here health care personnel) do not have a strong background in programming.

The second advantage is the easiness and the certainty to create a robust monitoring system for a relatively small scale problem domain such as, remote data acquisition. The advantage derives from LabVIEW's predefined virtual instruments for data acquisition and several kinds of communication protocols.

Paper IV: A highly automated documentation system: component design

Paper **IV** presents the modular structure of the ADS. The structure has been designed by combining methods used in VDF application design with the methods that are commonly used in designing applications based on a conventional PL. As a case example, the design process of a short distance component (SDC) is described. At that point, at which parts of the ADS had been prototyped, the top-down design method and VI prototyping were found to be usable. Nevertheless, it was also possible to note the possibility of future problems, such as the restriction in run-time program variability.

The main contribution of the paper is to present how the modularity of the VDFL and component design fits together. The paper also sets the basis for the research and implementation process described in Papers **V** and **VI**.

Paper V: Implementation of an automated documentation system with a visual data flow programming language

This paper is a continuation of the previous paper in that it introduces advantages and restrictions founded during the implementation of the ADS. The top-down design, combined with rapid prototyping, has been noted as a practical method for developing measurement components. However, using the top-down design method as the only design tool does not necessarily meet *all* defined requirements. With the top-down method, however, the user will not get any help how to manage the interaction between program components, for instance. Nor the method does not offer any glues for necessary types of data. In its own part this is important in designing program components and the input and output values they offer.

Restrictions have also been noted in a program's run-time variability and in its influence on the maintainability of the ADS. The paper introduces a theoretical solution entitled "*a dynamic case structure*" that exploits a property node with which to dynamically create the content of the case structure.

Neither the discovered restrictions nor the proposed solution have related research.

The contribution of the paper is to introduce the previously mentioned restrictions and challenges that appeared during the implementation of the ADS.

Paper VI: A monolithic program vs. modifiability: enhancing a visual data flow program with object-oriented techniques

As Paper **VI** explains, attaching a new measurement device into the ADS can be managed through visual object-oriented programming (VOOP) tools. The paper discusses the lack of program dynamics and the problem of monolithic program structure, as well as how these problems can be avoided with the use of data inheritance and data encapsulation. The VOOP tools have been tested in the implementation of the ADS's data acquisition (DAQ) component. The flexibility and modifiability of the DAQ component was enhanced because of the ability to create user defined data types. For example, defining a data type that is based on a measurement device ancestor

class made it possible to dynamically introduce and initialize only the necessary measurement objects.

Although data encapsulation and data inheritance have been described as usable tools against monolithic VDF program structure, the VOOP tools have not been found to be particularly effective for that purpose. The main reason for this is related to visual program structures that operate as functions. Paper **VI** presents a *dynamic call problem* which is an example of the challenge to manage some program parts to execute while other stay idle. The paper also presents how the use of control structures (here an iteration structure) easily makes the VDF program more monolithic.

The contribution of this paper is that it introduces the possibilities the VOOP tool can bring to VDF programming. The contribution also contains the representation of 1) the restrictions that can be avoided by using VOOP and 2) the restrictions of monolithic program structure that can not be avoided by using the VOOP.

Paper VII: Visual data flow languages with action systems

This paper is indirectly related to the ADS. The main idea of the paper relates to the question of whether the robustness of the ADS can be formally verified. The question is especially important when the ADS is used for patient care. In this paper, the basic concepts of VDFL is formalized with action systems. The mappings from the visual computational node, node composition, case structure and merge structure into to action systems are presented. The mapping has been discovered to be natural because of the similarity in the semantics of VDFL and action systems. The most important similarity can be found in parallel program execution, although it is important to take differences in execution into account. An executable action will not necessarily be executed at all in an action system, while in the VDFL an executable node is always executed. Another difference can be found in the naming of variables representing data arcs. However, these differences do not prevent or complicate the usage of actions systems in formalizing VDFLs. The results heavily encourage the continuation of work on reasoning and refining VDFL-based programs.

The contribution of this paper is that it presents the basic mapping from the VDFL into the action system formalism.

Paper VIII: Not-so-free data flow in a visual data flow programming language

This paper continues the discussion about the restricted ability to modify a VDF program during runtime. The paper introduces an attempt to visualize an optical communication network in LabVIEW. The visualization is based on the exploitation of the visual program *code* while the user interface is omitted from the review. The research contains work that has a twofold purpose:(1) to create a LabVIEW code similar to 4×4 optical torus and (2) to then visualize the flow of data tokens by exploiting LabVIEW's slow-motion execution ability.

The main problem was caused by the absence of a switch-alike VDF control mechanism able to route incoming data to the desired output. The requirement derived from the *partial dependency* of incoming data. Furthermore, it was noticed during the research that the absence of *null value* caused restrictions in implementation.

Paper **VIII** proposes three solutions to this problem. Each of the proposed solution simulates the functioning of the router in optical torus. A *dummy packets* solution uses empty arrays or empty string for enabling fluent flow of data. The two other solutions exploit an ability to monitor the content of data wires which requires the use of null values (which the recent version of LabVIEW does not have). An *event switch* solution is based on an event structure with an extended support of event rules. The solution also uses a property node for transferring data into the desired output. A *dynamic computational node* (DCN) is based on dynamic data terminals. According to which terminals are activated by new data, the DCN executes the corresponding program code.

It should be noted that wherever the DCN is used, it also forces every other node attached to the DCN's data wires to be dynamic. Therefore, a program consist of either the DCNs or basic nodes. The advantage of using the DCNs instead of variables can be seen in the visual code representation. In the DCN, the representation of program code is still more concrete than in the case of using variables. The program that consists of the DCNs can be viewed as a program that is made up of *program "variations"*. The "variation" of the program that is actually executed depends on which data wires are populated by new data tokens. What comes to the data flow paradigm,

the solution does not break the paradigm since program variation is only executable when all its inputs have new values.

This paper makes two contributions: (1) it presents the challenges and restrictions of VDFL discovered during the implementation of visualization system, and (2) it describes three solutions for solving the problem.

8 Conclusions and future work

Any concrete examples of how VDFL's restrictions emerge in practice are rarely discussed. Possible restrictions and problems in programming have been either knowingly or unknowingly avoided by exploiting global or local variables. In VDFL, the usage of variables fights strongly against the data flow paradigm and the variables also hide the data flow. In other words, if the restrictions and problems in VDF programming are typically avoided by using variables, this eliminates the great advantage of the VDFL, which is its support of parallelism.

A VDFL's restriction has been discovered in the communication between computational nodes by excluding the use of local and global variables, . Under the data flow paradigm, the nodes (including condition and iteration structures) operate as functions. Without variables, the runtime communication between nodes can be strongly restricted inside a "closed node", which makes it difficult to implement the communication. The previous communication problem again causes the program structure to be monolithic.

Problems that affect the program's variability were also found in the program dynamics. This study has presented examples of how the problem comes out in practice.

In order to avoid the monolithic program structure and to enhance the program dynamics, this study includes solution proposals, both in practice and in theory. Even though the monolithic program could not have been noticeably enhanced through the use of visual object-oriented tools, the exploitation of user defined data types and data encapsulation can be considered as a good way to improve VDFL applicability. The dynamic case structure, the dynamic computational node and the event switch structure are some solutions to the lack of program dynamics in VDFL.

Since the data flow computational model is a special case of Kahn's process networks (KPN) model [40, 45], similarities can be noticed in the

models' restrictions and enhancement proposals. The restriction that led to introduce the dynamic computational node for VDFLs, for instance, is similar to the Vrba's representation [47] of KPN's *limitation of determinism*. Vrba, as well as Lee et al. [46], states that KPN's limitations can be avoided with a *non-deterministic merge*. The use of non-deterministic merge in KPNs contributes to the use of dynamic computational node in VDFLs, because the behavior of the non-deterministic merge is similar to the behavior of dynamic computational node. In both, a node is executable when data appear on *any* of its inputs. Considering Vrba's proposals and this study, it seems natural to equip VDFL with structures that is similar to the behavior of KPN's non-deterministic merge.

The study also identified already widely stated advantages of VDFL. Rapid prototyping enables easy and fast program development. Furthermore, the top-down design method was a practical way of designing the hierarchical model of the measurement application. Designing small measurement systems, such as parts of the ADS, reveals the advantages of rapid prototyping and the top-down design method. However, restrictions can be identified here as well. More design tools and methods will be needed for the far-reaching and proper development of larger scale applications. This thesis argues that rapid prototyping and top-down design methods are inadequate for this purpose. Research into better design tools and methods for VDFLs can be seen as a fertile field for future research.

The current design methods raise questions about the robustness and effective functioning of the final application. Considering the use of a measurement system for patient care, there is no room for negative surprises. This study has introduced the use of formal methods in verification. The focus is set particularly on action systems and their ability to formalize VDFLs. Consequently, it has been noticed that action systems fit well for this purpose To the best of the author's knowledge, the formalization of VDFL with action systems has not previously been researched. The results strongly encouraged the continuing future use of the research.

In addition to the research fields of language improvements, better design methods, and the formalization, several other interesting research fields can be found among VDFLs. The demand driven approach of data flow could be an answer for a more dynamic VDF program; this is an idea that

has not been widely studied. Design methods represent another promising research area. The design methods can be enhanced if more attention is paid to improvements of VDFL. Furthermore, when object-oriented characteristics are combined with the VDFL, the design methods can also be partially borrowed from OO-programming. There is some on-going research in this area.

This thesis suggests that the development of VDFL is moving towards a language that combines the best characteristics of VDFL and control flow-based textual languages. The VOOP, for instance, is a good example of made improvements that does not break the data flow. This would make it possible to use more abstract structures than are currently available in VDFLs. At the same time, the research can offer a more comprehensible programming language than is represented by today's object-oriented PLs. As Meyer and Masterson stated in [67], "Properly coupled with object-oriented concepts, true visual programming could open the doors to a wider group of programmers, those who minds easily grasp the factory metaphor".

Bibliography

[1] E. Baroth and C. Hartsough, "Visual Programming in the real World," in *Visual Object-Oriented Programming*, M. Burnett, A. Goldberg, and T. Lewis, eds. (Manning Publications, 1995).

[2] W. Johnston, P. Hanna, and R. Millar, "Advances in Dataflow Programming Languages," *ACM Computing Surveys* **36,** 1–34 (2004).

[3] D. Hils, "Visual Languages and Computing Survey: Data flow Visual programming Languages," *Visual Languages and Computing* **3,** 69–101 (1992).

[4] W. Ackerman, "Data Flow Languages," *Computer* **15,** 15–25 (1982).

[5] S. Bhattacharyya, G. Brebner, J. W. Janneck, J. Eker, C. von Platen, M. Mattavelli, and M. Raulet, "OpenDF: a dataflow toolset for reconfigurable hardware and multicore systems," *SIGARCH Computer Architecture News* **36,** 29–35 (2008).

[6] J. B. Dennis, "First version of a data flow procedure language," in *Programming Symposium, Proceedings Colloque sur la Programmation* (1974), pp. 362–376.

[7] P. R. Kosinski, "A data flow language for operating systems programming," *SIGPLAN Not.* **8,** 89–94 (1973).

[8] M. Burnett, C. Bogart, J. Cao, V. Grigoreanu, T. Kulesza, and J. Lawrance, "End-user software engineering and distributed cognition," in *SEEUP '09: Proceedings of the 2009 ICSE Workshop on Software Engineering Foundations for End User Programming* (2009), pp. 1–7.

[9] D. Grimaldi and S. Rapuano, "Hardware and software to design virtual laboratory for education in instrumentation and measurement," *Measurement* **42,** 485–493 (2009).

[10] M. Kindborg and K. McGee, "Visual programming with analogical representations: inspirations from a semiotic analysis of comics," *Visual Languages and Computing* **18,** 99—125 (2007).

[11] G. Lewis, D. Smith, L. Bass, and B. Myers, "Report of the workshop on software engineering foundations for end-user programming," *SIGSOFT Software Engineering Notes* **34,** 51–54 (2009).

[12] M. Petre, "Cognitive dimensions beyond the notation," *Visual Languages and Computing* **17,** 292—301 (2006).

[13] Y.-Y. Sui, J. Lin, and X. Z. Zhang, "An automated refactoring tool for dataflow visual programming language," *SIGPLAN Not.* **43,** 21–28 (2008).

[14] L. Nailu, L. Yuegang, and X. Peiyu, "A real-time simulation system of wind power based on LabVIEW DSC module and Matlab/Simulink," in *Proceedings of the 9th International Conference on Electronic Measurement & Instruments, ICEMI* (2009), pp. 547–552.

[15] S. Putta, V. Vaidyanathan, and J. Chung, "Development and testing of a nodal resistance measurement (NRM) system for composite structures," *Measurement* **41,** 763—773 (2008).

[16] C. Wagner, S. Armenta, and B. Lendl, "Developing automated analytical methods for scientific environments using LabVIEW," *Talanta* **80,** 1081–1087 (2009).

[17] S. Gadgil, D. D. Verma, M. S. Panse, and K. Tuckley, "Sea State Monitoring HF Radar Controller Using Reconfigurable LabVIEW FPGA," in *ACT '09: International Conference on Advances in Computing, Control, & Telecommunication Technologies* (2009), pp. 395–397.

[18] M. E. Jackson and J. W. Gnadt, "Numerical simulation of nonlinear feedback model saccade generation circuit implemented in the LabVIEW graphical programming language," *Neuroscience methods* **87,** 137–145 (1999).

[19] J. Li and W. Zheng, "Research on Three-Dimensional Sport Testing and Analyzing System Based on LabVIEW," in *Proceedings of the International Symposium on Knowledge Acquisition and Modeling* (2008), pp. 750–753.

[20] K. Komala, M. Kurian, and A. Shivannavar, "Real time access & control of ECG signals using lab view based web browser," in *Anti-counterfeiting, Security, and Identification in Communication, 2009. ASID 2009. 3rd International Conference on* (2009), pp. 406–409.

[21] D. Taihang, S. Yanli, H. Xiangbin, and M. Qing, "The Technology of Contact Image Acquisition and Progressing in Relay Electrical Endurance Test," in *Proceedings of the Second International Conference on Intelligent Networks and Intelligent Systems* (2009), pp. 330–333.

[22] G. Wang, S. Jiao, and H. Song, "Mine Pump Comprehensive Performance Testing System Based on LabVIEW," in *ICMTMA '09: Proceedings of the 2009 International Conference on Measuring Technology and Mechatronics Automation* (2009), pp. 300–303.

[23] X. Xuejun, X. Ping, Y. Sheng, and L. Ping, "Real-time Digital Simulation of Control System with LabVIEW Simulation Interface Toolkit," in *Proceedings of the Control Conference, CCC. Chinese* (2007), pp. 318–322.

[24] National Instruments, "LabVIEW," url: http://www.ni.com/labview (cited April 2009).

[25] M. Chen, "Object oriented programming in LabVIEW for acquisition and control systems at the Aerodynamics Laboratory of the National Research Council of Canada," in *International Congress on Instrumentation in Aerospace Simulation Facilities* (2007), pp. 1–6.

[26] G. Faraco and L. Gabriele, "Using LabVIEW for applying mathematical models in representing phenomena," *Comput. Educ.* **49,** 856–872 (2007).

[27] K. Whitley, L. Novick, and D. Fisher, "Evidence in Favor of Visual Representation for the Dataflow Paradigm: An Experiment Testing LabVIEW's Comprehensibility," *Human-Computer Studies* **64,** 281–303 (2005).

[28] K. N. Whitley and A. F. Blackwell, "Visual programming: the outlook from academia and industry," in *ESP '97: Papers presented at the seventh workshop on Empirical studies of programmers* (1997), pp. 180–208.

[29] J. Backus, "Can Programming Languages Be liberated from the von Neumann Style? A Functional Style and its Algebra of Programs," *Communications of the ACM* **21,** 613–641 (1978).

[30] R. A. Iannucci, "Toward a dataflow/von Neumann hybrid architecture," in *ISCA '88: Proceedings of the 15th Annual International Symposium on Computer architecture* (1988), pp. 131–140.

[31] W. Najjar, E. Lee, and G. Gao, "Advances in the data flow computational model," *Parallel Computing* **25,** 1907–1929 (1999).

[32] Arvind and R. A. Iannucci, "A critique of multiprocessing von Neumann style," in *ISCA '83: Proceedings of the 10th annual international symposium on Computer architecture* (1983), pp. 426–436.

[33] B. G. Ryder, M. L. Soffa, and M. Burnett, "The impact of software engineering research on modern progamming languages," *ACM Trans. Softw. Eng. Methodol.* **14,** 431–477 (2005).

[34] A. L. Davis and R. M. Keller, "Data Flow Program Graphs," *Computer* **15,** 26–41 (1982).

[35] D. D. Chamberlin, "The "single-assignment" approach to parallel processing," in *AFIPS '72 (Spring): Proceedings of the May 16-18, 1972, spring joint computer conference* (1971), pp. 263–269.

[36] P. G. Whiting and R. S. Pascoe, "A History of Data-Flow Languages," *IEEE Ann. Hist. Comput.* **16,** 38–59 (1994).

[37] A. Ambler and M. Burnett, "Visual forms of iteration that preserve single assignment," *Visual Languages and Computing* **12,** 123–125 (2001).

[38] R. Bird and P. Wadler, *Introduction to functional programming* (Prentice Hall, 1998).

[39] A. L. Davis and S. A. Lowder, "A Sample Management application program in a graphical data-driven programming language," *Digest of Papers Compcon Spring* 241–271 (1974).

[40] G. Kahn, "The semantics of a simple language for parallel programming," in *Proceedings of the IFIP Congress 74* (1974), pp. 471–475.

[41] J. A. Sharp, "Some thoughts on data flow architectures," *SIGARCH Comput. Archit. News* **8,** 11–21 (1980).

[42] J. B. Dennis and D. P. Misunas, "A preliminary architecture for a basic data-flow processor," in *ISCA '75: Proceedings of the 2nd annual symposium on Computer architecture* (1975), pp. 126–132.

[43] Y. Litvin, "Parallel evolution programming language for data flow machines," *SIGPLAN Not.* **17,** 50–58 (1982).

[44] A. L. Ambler and M. M. Burnett, "Visual languages and the conflict between single assignment and iteration," in *IEEE Workshop on Visual Languages* (1989), pp. 138 –143.

[45] E. A. Lee and A. Sangiovanni-Vincentelli, "Comparing models of computation," in *Proceedings of the 1996 IEEE/ACM international conference on Computer-aided design*, ICCAD '96 (1996), pp. 234–241.

[46] E. Lee and T. Parks, "Dataflow process networks," *Proceedings of the IEEE* **83,** 773 –801 (1995).

[47] Z. Verba, *Implementation and performance aspects of Kahn process networks*, PhD thesis (University of Oslo, 2009).

[48] J. D. Kiper, E. Howard, and C. Ames, "Criteria for Evaluation of Visual Programming Languages," *Visual Languages and Computing* **8,** 175–192 (1997).

[49] S.-K. Chang, "Visual languages: a tutorial and survey," in *Psychology. Selected contributions on Visualization in programming. 5th Interdisciplinary Workshop in Informatics* (1987), pp. 1–23.

[50] M. Burnett, "Visual Programming," in *Encyclopedia of Electrical and Electronics Engineering* (1999).

[51] G. Costagliola, A. Delucia, S. Orefice, and G. Polese, "A Classification framework to support the design of visual languages," *Visual Languages and Computing* **13,** 573–600 (2002).

[52] M. Mosconi and M. Porta, "Iteration constructs in data-flow visual programming languages," *Computer Languages* **26,** 67–104 (2000).

[53] M. Burnett, "Graphical Definitions: Expanding Spreadsheet Languages through Direct Manipulation and Gestures," *Computer-Human Interaction* **5,** 1–33 (1998).

[54] A. Kay, "Computer Software," *Scientific American* **5,** 53–59 (1984).

[55] D. Kurlander, "Chimera: Example-based graphical editing," in *Watch What I Do: Programming by Demonstration*, A. Cypher, ed. (MIT Press, 1993).

[56] B. A. Myers, "Visual programming, programming by example, and program visualization: a taxonomy," *SIGCHI Bull.* **17,** 59–66 (1986).

[57] M. Burnett, A. Goldberg, and T. Lewis, "What is visual object-oriented programming?," in *Visual object oriented programming*, M. Burnett, A. Goldberg, and T. Lewis, eds. (Manning Publications, 1995).

[58] B. Shneiderman, "Direct manipulation: A Step Beyond Programming Languages," *Computer* **16,** 57—69 (1983).

[59] S. Tanimoto, "VIVA:a visual language for image processing," *Visual Languages and Computing* **1,** 127–139 (1990).

[60] A. F. Blackwell, "Metacognitive theories of visual programming: What do we think we are doing?," in *Proceedings IEEE Symposium on Visual Languages* (IEEE Computer Society, 1996), pp. 240–246.

[61] I. Bell, J. Falcon, J. Limroth, and K. Robinson, "Integration of hardware into the LabVIEW environment for rapid prototyping and the development of control design applications," in *UKACC Control 2004 Mini Symposia* (2004), pp. 79–81.

[62] D. J. Parish and P. Reeves-Hardcastle, "Rapid prototyping using the LabVIEW environment," in *AUTOTESTCON '96, 'Test Technology and Commercialization'. Conference Record* (1996), pp. 235–238.

[63] R. Jamal and L. Wenzel, "The applicability of the visual programming language LabVIEW to large real-world applications," in *Proceedings of the 11th symposium of Visual Languages* (1995), pp. 99–106.

[64] K. Whitley, "Visual programming languages and the empirical evidence for and against," *Visual Languages and Computing* **8,** 109–142 (1997).

[65] M. Burnett, M. Baker, C. Bohus, P. Carlson, S. Yang, and P. Van Zee, "Scaling up visual programming languages," *Computer* **28,** 45 –54 (1995).

[66] B. Shizuki, M. Toyoda, E. Shibayama, and S. Takahashi, "Smart browsing among multiple aspects of data-flow visual program execution, using visual patterns and multi-focus fisheye views," *Visual Languages and Computing* **11,** 529–548 (2000).

[67] M. Meyer and T. Masterson, "Towards a better visual programming language: critiquing prograph's control structures," *Journal of computing sciences in colleges* **15,** 181–193 (2000).

[68] X. Shen, J. Gu, X. Dong, and G. Wang, "A Method to Solve Small-Screen and Scaling-up Problems in VPLs with Fisheye Views," in *Proceedings of the International Symposium on Information Science and Engieering, ISISE*, Vol. 1 (2008), pp. 93 –97.

[69] Y. Sui, L. Pang, J. Lin, and X. Zhang, "Dataflow Visual Programming Language Debugger Supported by Fisheye View," in *ICYCS '08: Proceedings of the 2008 The 9th International Conference for Young Computer Scientists* (2008), pp. 1024–1029.

[70] P. Gross and K. Powers, "Evaluating assessments of novice programming environments," in *ICER '05: Proceedings of the first international workshop on Computing education research* (2005), pp. 99–110.

[71] C. Hundhausen and J. LeeBrown, "An experimental study of the impact of visual semantic feedback on novice programming," *Visual Languages and Computing* **18,** 537–559 (2007).

[72] C. Kelleher and R. Pausch, "Lowering the barriers to programming: A taxonomy of programming environments and languages for novice programmers," *ACM Comput. Surv.* **37,** 83–137 (2005).

[73] H. Lieberman, "Programming by example (introduction)," *Commun. ACM* **43,** 72–74 (2000).

[74] B. A. Myers, "Creating user interfaces using programming by example, visual programming, and constraints," *ACM Trans. Program. Lang. Syst.* **12,** 143–177 (1990).

[75] B. A. Myers, R. McDaniel, and D. Wolber, "Programming by example: intelligence in demonstrational interfaces," *Commun. ACM* **43,** 82–89 (2000).

[76] D. C. Smith, A. Cypher, and L. Tesler, "Programming by example: novice programming comes of age," *Commun. ACM* **43,** 75–81 (2000).

[77] R. St. Amant, H. Lieberman, R. Potter, and L. Zettlemoyer, "Programming by example: visual generalization in programming by example," *Commun. ACM* **43,** 107–114 (2000).

[78] T. Green and M. Petre, "Usability Analysis of Visual Programming Environments: A Cognitive Dimensions Framework," *Visual Languages and Computing* **7,** 131—174 (1996).

[79] C. Cook, M. Burnett, and D. Boom, "A bug's eye view of immediate visual feedback in direct-manipulation programming systems," in *ESP '97: Papers presented at the seventh workshop on Empirical studies of programmers* (1997), pp. 20–41.

[80] E. M. Wilcox, J. W. Atwood, M. M. Burnett, J. J. Cadiz, and C. R. Cook, "Does continuous visual feedback aid debugging in direct-manipulation programming systems?," in *CHI '97: Proceedings of the SIGCHI conference on Human factors in computing systems* (1997), pp. 258–265.

[81] M. Petre, "Why looking isn't always seeing: readership skills and graphical programming," *Commun. ACM* **38,** 33–44 (1995).

[82] S. Clarke, "Evaluating a new programming language," in *13th Workshop of the Psychology of Programming Interest Group* (2001), pp. 275–289.

[83] J. Dagit, J. Lawrance, C. Neumann, M. Burnett, R. Metoyer, and S. Adams, "Using cognitive dimensions: Advice from the trenches," *Visual languages and computing* **17,** 302–327 (2006).

[84] D. Edge and A. Blackwell, "Correlates of the cognitive dimensions for tangible user interface," *Visual Languages and Computing* **17,** 366—394 (2006).

[85] T. Green, M. Petre, A. Blandford, L. Church, and C. Roast, "Cognitive dimensions: Achievements, new directions, and open questions," *Visual Languages and Computing* **17,** 328—365 (2006).

[86] C. Hundhausen, "Using end-user next term visualization environments to mediate conversations: a Communicative Dimensions framework," *Visual Languages and Computing* **16,** 153–185 (2005).

[87] M. Auguston and A. Delgado, "Iterative constructs in the visual data flow language," in *Visual Languages, 1997. Proceedings. 1997 IEEE Symposium on* (1997), pp. 152 –159.

[88] M. Mosconi and M. Porta, "A Data-Flow Visual Approach to Symbolic Computing: Implementing a Production-Rule-Based Programming System through a General-Purpose Data-Flow VL," in *VL '00: Proceedings of the 2000 IEEE International Symposium on Visual Languages (VL'00)* (2000), p. 83.

[89] H. Randriamparany and B. Ibrahim, "Seamless Integration of Control Flow and Data Flow in a Visual Language," in *AICCSA '01: Proceedings of the ACS/IEEE International Conference on Computer Systems and Applications* (2001), pp. 428–434.

[90] E. Ghittori, M. Mosconi, and M. Porta, "Designing New Programming Constructs in a Data Flow VL," in *VL '98: Proceedings of the IEEE Symposium on Visual Languages* (1998), p. 78.

[91] P. R. Kosinski, "Mathematical semantics and data flow programming," in *POPL '76: Proceedings of the 3rd ACM SIGACT-SIGPLAN symposium on Principles on programming languages* (1976), pp. 175–184.

[92] J. B. Dennis, G. A. Boughton, and C. K. Leung, "Building blocks for data flow prototypes," in *ISCA '80: Proceedings of the 7th annual symposium on Computer Architecture* (1980), pp. 1–8.

[93] B. Guo, J. Zhang, and X. Nie, "Application of LabVIEW for Hydraulic Automatic Test System," in *IIS '09: Proceedings of the 2009 International Conference on Industrial and Information Systems* (2009), pp. 348–351.

[94] J. M. Ramírez, P. Gómez-Gil, and F. L. Larios, "A Robot-vision System for Autonomous Vehicle Navigation with Fuzzy-logic Control

using LabVIEW," in *CERMA '07: Proceedings of the Electronics, Robotics and Automotive Mechanics Conference* (2007), pp. 295–302.

[95] P. Hosek, T. Prykäri, E. Alarousu, and R. Myllylä, "Application of LabVIEW: Complex Software Controlling of System for Optical Coherence Tomography ," *Association for Laboratory Automation* **14,** 59–68 (2009).

[96] M. Benghanem, "A low cost wireless data acquisition system for weather station monitoring," *Renewable Energy* **35,** 862–872 (2009).

[97] R. A. Dielenberg, P. Halasz, and T. A. Day, "A method for tracking rats in a complex and completely dark environment using computerized video analysis," *Neuroscience methods* 279–286 (2006).

[98] J.-H. Horng, "Hybrid MATLAB and LabVIEW with neural network to implement a SCADA system of AC servo motor," *Advances in Engineering Software* **39,** 149–155 (207).

[99] J.-Q. Ni and A. J. Hebert, "An on-site computer system for comprehensive agricultural air quality research," *Computers and Electronics in Agriculture (article in press)* (2010).

[100] K. Prema, N. S. Kumar, and K. Sunitha, "Online temperature control based on virtual instrumentation," in *Proceedings of the International Conference on Control, Automation, Communication and Energy Conservation, INCACEC* (2009), pp. 1 –4.

[101] H. Qiong and F. Shidong, "Development of Pipeline Leak Detection System based on LabVIEW," in *Proceedings of the International Symposium on Knowledge Acquisition and Modeling, KAM* (2008), pp. 671–674.

[102] M. Zulkifli, S. Harun, K. Thambiratnam, and H. Ahmad, "Self-Calibrating Automated Characterization System for Depressed Cladding EDFA Applications Using LabVIEW Software With GPIB," *Instrumentation and Measurement, IEEE Transactions on* **57,** 2677–2681 (2008).

[103] F. C. Alegria, E. Martinho, and F. Almeida, "Measuring soil contamination with the time domain induced polarization method using LabVIEW," *Measurement* **42,** 1082—1091 (2009).

[104] Q. Tang, Z. Teng, S. Guo, and Y. Wang, "Design of Power Quality Monitoring System Based on LabVIEW," in *Proceeding of the International Conference on Measuring Technology and Mechatronics Automation*, Vol. 1 (2009), pp. 292–295.

[105] M. Palit, N. Bhattacharyya, S. Sarkar, A. Dutta, P. Dutta, B. Tudu, and R. Bandyopadhyay, "Virtual Instrumentation Based Voltammetric Electronic Tongue for Classification of Black Tea," in *Proceedings of the third international Conference on Industrial and Information Systems, ICIIS* (2008), pp. 1–6.

[106] T. G. Sparkman, "Lessons Learned Applying Software Engineering Principles to Visual Programming Language Application Development," in *COMPSAC '99: 23rd International Computer Software and Applications Conference* (1999), p. 416.

[107] A. Lukindo, "LabVIEW Queued State Machine Architecture," url: http://expressionflow.com/2007/10/01/labview-queued-state-machine-architecture/ (cited Oct 2007).

[108] T. Maila, "Worker pool - a design pattern for parallel task execution in LabVIEW," url: http://expressionflow.com/2009/11/04/worker-pool/ (cited Feb 2010).

[109] M. Roe, D. Brandon, P. McDowell, and B. Bourgeios, "A flexible client/server application for robotic control," in *ACM-SE 43: Proceedings of the 43rd annual Southeast regional conference* (2005), pp. 182–187.

[110] M. Toyoda, B. Shizuki, S. Takahashi, S. Matsuoka, and E. Shibayama, "Supporting Design Patterns in a Visual Parallel Data-flow Programming Environment," in *VL '97: Proceedings of the 1997 IEEE Symposium on Visual Languages (VL '97)* (1997), p. 76.

[111] M. Burnett, "Software Engineering For Visual Programming Languages," in *Handbook of Software Engineering and Knowledge Engineering* (World Scientific Publishing Company, 2001).

[112] N. Kehtarnavaz and C. Gope, "DSP System Design Using Labview and Simulink: A Comparative Evaluation," in *Proceedings of the International Conference on Acoustics, Speech and Signal Processing, ICASSP 2006*, Vol. 2 (2006), pp. 985 – 988.

[113] P. Cox, S. Gauvin, and A. Rau-Chaplin, "Adding parallelism to visual data flow programs," in *SoftVis '05: Proceedings of the 2005 ACM symposium on Software visualization* (2005), pp. 135–144.

[114] L. Cranor and A. Apte, "Programs worth one thousand words: visual languages bring programming to the masses," *Crossroads* **1,** 16–18 (1994).

[115] J. Webber and P. Lee, "Visual, Object-oriented Development of Parallel Applications," *Visual Languages and Computing* **12,** 145–161 (2001).

[116] D. D. Hils, "Datavis: a visual programming language for scientific visualization," in *CSC '91: Proceedings of the 19th annual conference on Computer Science* (1991), pp. 439–448.

[117] A. Fukunaga, W. Pree, and T. D. Kimura, "Functions as objects in a data flow based visual language," in *CSC '93: Proceedings of the 1993 ACM conference on Computer science* (1993), pp. 215–220.

[118] A. Fukunaga, T. D. Kimura, and W. Pree, "Object-oriented development of a data flow visual language system," in *Proceedings of the IEEE Symposium on Visual Languages* (1993), pp. 134 –141.

[119] D. Dearman, A. Cox, and M. Fisher, "Adding control-flow to a visual data-flow representation," in *Proceedings of the 13th International Workshop on Program Comprehension, IWPC* (2005), pp. 297 – 306.

[120] M. Erwig, "Abstract syntax and Semantics of Visual Languages," *Visual Languages and Computing* **9,** 461—483 (1998).

[121] D. M. Gee, "Formal specification of visual languages," *Information and software technology* **40,** 359–367 (1998).

[122] E. J. Golin and S. P. Reiss, "The Specification of Visual Language Syntax," *Visual Language and Computing* **1,** 141–157 (1990).

[123] K. M. Kavi, B. P. Buckles, and U. N. Bhat, "A Formal Definition of Data Flow Graph Models," *IEEE Trans. Comput.* **35,** 940–948 (1986).

[124] S. Sumathi and P. Surekha, *LabVIEW based Advanced Instrumentation Systems* (Springer, 2007).

[125] J. B. Olansen and E. Rosow, *Virtual Bio-Instrumentation; Biomedical, clinical, and heathcare applications in LabVIEW* (Prentice Hall, 2002).

[126] National Instruments, "Using Local and Global Variables Carefully," url: http://zone.ni.com/reference/en-XX/help/371361E-01/lvconcepts/using_ local_ and _ global/ (cited January 2011).

[127] National Instruments, "LabVIEW Object-Oriented Programming FAQ," url: http://zone.ni.com/devzone/cda/tut/p/id/3573 (cited April 2010).

[128] L. Xinmei and Z. Ziyu, "A Study of Transient Temperature Measuring System Based on LabVIEW for Droplets," in *CSSE '08: Proceedings of the 2008 International Conference on Computer Science and Software Engineering* (2008), pp. 476–480.

[129] A. Poursaee and W. J. Weiss, "An automated electrical monitoring system (AEMS) to assess property development in concrete," *Automation in Construction* **19,** 485—490 (2010).

[130] Y. Chu and A. Ganz, "WISTA: a wireless telemedicine system for disaster patient care," *Mob. Netw. Appl.* **12,** 201–214 (2007).

[131] C. L. Bryant and N. J. Gandhi, "Real-time data acquisition and control system for the measurement of motor and neural data," *Journal of Neuroscience methods* **142,** 193–200 (2005).

[132] P. Ponce, F. Ramirez, and V. Medina, "A novel neuro-fuzzy controller genetically enhanced using LabVIEW," in *Proceedings of the 34th Annual Conference of IEEE on Industrial Electronics* (2008), pp. 1559–1565.

[133] C. Pornpatkul and T. Suksri, "Decentralized fuzzy logic controller for TITO coupled-tank process," in *Proceedings of the ICROS-SICE International Joint Conference 2009* (2009), pp. 2862–2866.

[134] A. Tiwari, P. Ballal, and F. L. Lewis, "Energy-efficient wireless sensor network design and implementation for condition-based maintenance," *ACM Trans. Sen. Netw.* **3,** 1–23 (2007).

[135] T. Kim and Y. Chung, "Efficient hardware IP control and simulation with LabVIEW," in *ASIC, 2009. ASICON '09. IEEE 8th International Conference on* (2009), pp. 914–916.

[136] R. Gutiérrez-Castrejón and M. Duelk, "Using LabVIEW for advanced nonlinear optoelectronic device simulations in high-speed optical communications," *Computer Physics Communications* **174,** 431–440 (2006).

[137] K. Kim and Rizwan-uddin, "A web-based nuclear simulator using RELAP5 and LabVIEW," *Nuclear Engineering and Design* **237,** 1185–1194 (2007).

[138] A. Y. Al-Zoubi, S. Jeschke, N. M. Natho, J. Nsour, and O. F. Pfeiffer, "Integration of an online digital logic design lab for IT education," in *SIGITE '08: Proceedings of the 9th ACM SIGITE conference on Information technology education* (2008), pp. 237–242.

[139] F. Sandu, P. Nicolae, E. Kayafas, and S. A. Moraru, "LabVIEW-based remote and mobile access to real and emulated experiments in electronics," in *PETRA '08: Proceedings of the 1st international conference on Pervasive Technologies Related to Assistive Environments* (2008), pp. 1–7.

[140] F. Coito and L. B. Palma, "A remote laboratory environment for blended learning," in *PETRA '08: Proceedings of the 1st interna-*

tional conference on Pervasive Technologies Related to Assistive Environments (2008), pp. 1–4.

[141] C. Steidley and R. Bachnak, "Software and hardware development for a virtual laboratory," *J. Comput. Small Coll.* **20,** 200–206 (2005).

[142] L. Peretto, S. Rapuano, M. Riccio, and D. Bonatti, "Distance learning of electronic measurement by means of measurement set-up models," *Measurement* **41,** 274—283 (2008).

[143] N. K. Swain and M. Swain, "Design and development of computer networking modules using virtual instruments and object oriented programming," in *CITC5 '04: Proceedings of the 5th conference on Information technology education* (2004), pp. 270–270.

[144] Lego, "Lego MINDSTORMS," url: http://mindstorms.lego.com/en-us/default.aspx (cited November 2009).

[145] G. J. Ferrer, "Using Lego Mindstorms NXT in the classroom," *Journal of Computing Sciences Colleges* **23,** 153–153 (2008).

[146] M. A. Garcia and H. P. Mc-Neill, "Learning how to develop software using the toy LEGO Mindstorms," *SIGCSE Bull.* **34,** 239–239 (2002).

[147] S. H. Kim and J. W. Jeon, "Programming LEGO Mindstorms NXT with visual programming," in *Proceedings of the international conference on control, automation and systems* (2007), pp. 2468–2472.

[148] S. H. Kim and J. W. Jeon, "Using visual programming kit and LEGO Mindstorms: An introduction to embedded system," in *Proceedings of the international conference on industrial technology* (2008), pp. 1–6.

[149] D. E. Stevenson and J. D. Schwarzmeier, "Building an autonomous vehicle by integrating lego mindstorms and a web cam," in *SIGCSE '07: Proceedings of the 38th SIGCSE technical symposium on Computer science education* (2007), pp. 165–169.

[150] W. I. McWhorter and B. C. O'Connor, "Do LEGO® Mindstorms® motivate students in CS1?," in *SIGCSE '09: Proceedings of the 40th ACM technical symposium on Computer science education* (2009), pp. 438–442.

[151] A. B. Williams, "The qualitative impact of using LEGO MINDSTORMS robot to teach computer engineering," *IEEE Transactions on Education* **46,** 206–206 (2003).

[152] S. Yang and V. Ajjarapu, "Web-based speed control of induction motor with inverter dead-time compensation," in *SCSC: Proceedings of the 2007 summer computer simulation conference* (2007), pp. 141–148.

[153] J. Kalomiros and J. Lygouras, "Design and evaluation of a hardware/software FPGA-based system for fast image processing," *Microprocessors and Microsystems* **32,** 95–106 (2007).

[154] R. Bitter, T. Mohiuddin, and M. Nawrocki, *LabVIEW advanced programming techniques* (CRC-Press, 2001).

[155] D. Ingalls, S. Wallace, Y.-Y. Chow, F. Ludolph, and K. Doyle, "Fabrik: a visual programming environment," in *OOPSLA '88: Conference proceedings on Object-oriented programming systems, languages and applications* (1988), pp. 176–190.

[156] F. Ludolph, Y.-Y. Chow, D. Ingalls, S. Wallace, and K. Doyle, "The Fabrik programming environment," in *Proceedings of the IEEE Workshop on Visual Languages* (1988), pp. 222–230.

[157] M. Hirakawa, M. Tanaka, and T. Ichikawa, "An Iconic Programming System, HI-VISUAL," *IEEE Trans. Softw. Eng.* **16,** 1178–1184 (1990).

[158] M. Young, D. Argiro, and S. Kubica, "Cantata: a visual programming environment for the Khoros system," *ACM SIGGRAPH Computer Graphics* **29,** 22–24 (1995).

[159] S. Matwin and T. Pietrzykowski, "PROGRAPH: a preliminary report," *Comput. Lang.* **10,** 91–126 (1985).

[160] R. Angus and T. Hulbert, *VEE Pro: Practical Graphical Programming* (Springer-Verlag, 2005).

[161] M. Klinger, "Reusable test executive and test programs methodology and implementation comparison between HP VEE and LabVIEW," in *Proceedings of the AUTOTESTCON '99.* (1999), pp. 305–312.

[162] C. Elliott, V. Vijayakumar, W. Zink, and R. Hansen, "National Instruments LabVIEW: A Programming Environment for Laboratory Automation and Measurement," *Journal of the Association for Laboratory Automation* **12,** 17–24 (2007).

[163] C. Kalkman, "LabVIEW: a software system for data acquisition, data analysis, and instrument control," *Clinical monitoring and computing* **11,** 51–58 (1995).

[164] S. Clerici, C. Zoltan, and G. Prestigiacomo, "NiMoToons: a Totally graphic workbench for program tuning and experimentation," *Electronic Notes in Theoretical Computer Science* **258,** 93–107 (2009).

[165] Y. Kollet and T. J. Smedley, "Message-Flow Programming in PdaGraph," in *VLHCC '04: Proceedings of the 2004 IEEE Symposium on Visual Languages - Human Centric Computing* (2004), pp. 229–232.

[166] Microsoft, "VPL introduction," Website (2009), `http://msdn.microsoft.com/en-us/library/bb483088.aspx`.

[167] J. Reichardt, "Two-dimensional C++," in *SoftVis '06: Proceedings of the 2006 ACM symposium on Software visualization* (2006), pp. 191–192.

[168] R. J. Gran, *Numerical Computing with Simulink, Volume I: Creating Simulations* (Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 2007).

[169] A. Leff and J. T. Rayfield, "Relational Blocks: A Visual Dataflow Language for Relational Web-Applications," in *Proceedings of the IEEE Symposium on Visual Languages and Human-Centric Computing, VL/HCC* (2007), pp. 205 –208.

[170] P. Baldwin, S. Kohli, E. Lee, X. Liu, and Y. Zhao, "Modeling of sensor nets in Ptolemy II," in *Proceedings of the 3rd international symposium on Information processing in sensor networks* (2004), pp. 359–368.

[171] E. A. Lee, X. Liu, and S. Neuendorffer, "Classes and inheritance in actor-oriented design," *ACM Trans. Embed. Comput. Syst.* **8,** 1–26 (2009).

[172] N. Joncheere, D. Deridder, R. Van Der Straeten, and V. Jonckers, "A Framework for Advanced Modularization and Data Flow in Workflow Systems," *Lecture Notes in Computer Science* **5364,** 592–598 (2008).

[173] D. Liu, J. Wang, S. C. F. Chan, J. Sun, and L. Zhang, "Modeling workflow processes with colored Petri nets," *Computers in Industry* **49,** 267 – 281 (2002).

[174] K. Salimifard and M. Wright, "Petri net-based modelling of workflow systems: An overview," *European Journal of Operational Research* **134,** 664 – 676 (2001).

[175] O. Consortium, "OASIS Web Services Business Process Execution Language (WSBPEL)," url: oasis-open.org (cited November 2010).

[176] T. Oinn, M. Addis, J. Ferris, D. Marvin, M. Senger, M. Greenwood, T. Carver, K. Glover, M. R. Pocock, A. Wipat, and P. Li, "Taverna: a tool for the composition and enactment of bioinformatics workflows," *Bioinformatics* **20,** 3045–3054 (2004).

[177] D. Turi, P. Missier, C. Goble, D. De Roure, and T. Oinn, "Taverna Workflows: Syntax and Semantics," in *e-Science and Grid Computing, IEEE International Conference on* (2007), pp. 441 –448.

[178] E. Elmroth, F. Hernández, and J. Tordsson, "Three fundamental dimensions of scientific workflow interoperability: Model of computation, language, and execution environment," *Future Generation Computer Systems* **26,** 245 – 256 (2010).

Maija Marttila-Kontio

Visual Data Flow Programming Languages

Challenges and Opportunities

This thesis introduces the challenges and opportunities that visual data flow programming language (VDFL) presents to the field of software development.

UNIVERSITY OF
EASTERN FINLAND

Publications of the University of Eastern Finland
Dissertations in Forestry and Natural Sciences

ISBN 978-952-61-0417-1