Software Synthesis from Dataflow Models for G and LabVIEW™

Hugo A. Andrade
Scott Kovner

Department of Electrical and Computer Engineering
University of Texas at Austin
Austin, TX 78712
andrade@mail.utexas.edu
kovner@mail.utexas.edu

Embedded System Software (EE382C)
Literature Survey

March 16th, 1998

Abstract

The "G" programming language, as implemented in the National Instruments product "LabVIEW™", allows the user to describe a program with a dataflow representation. Our goal is to apply the techniques and concepts of the current dataflow research towards the adaptation of G as an embedded software development tool. LabVIEW™ is dominant in the instrumentation industry. As the instrumentation industry makes use of more embedded systems, it becomes practical to consider extending LabVIEW™'s and G's functionality to target embedded systems.

Formally, G is a homogeneous, multidimensional, dynamic dataflow language. G uses "structured dataflow" semantics to specify high level concepts (e.g. loops, conditional control flow, etc.) instead of using low level actors and feedback. We compare G to other models of computation, such as cyclostatic dataflow, dynamic dataflow, and process networks. In particular, we look for what we can learn from these models to apply to G.

This survey is a launching point for discussing possible changes to G. In the future, we will discuss what extensions may be necessary for G to be more useful for representing some of these models of computation. We will also discuss semantic and syntactic restrictions to G that may be helpful when using G to describe a particular computational model.

1 Introduction

The use of dataflow programming tools for system prototyping and development predates some of the recent work in compiling and scheduling dataflow graphs. For example, one popular dataflow language tool called LabVIEW™ was released in 1986, but much of the work on targeting general purpose computer architectures with dataflow has been published during the 1990's. In this literature survey, we will cover some of these recent developments and discuss how LabVIEW™ may be augmented to take advantage of these new developments.

2 LabVIEW™and G Background

LabVIEW™ (*Lab*oratory *V*irtual *I*nstrument *E*ngineering *W*orkbench) is a graphical application development environment (ADE) developed by National Instruments Corporation for the Data Acquisition (DAQ), Test and Measurement (T&M) and the Industrial Automation (IA) markets. It was originally developed in the early 1980's and is currently in its fifth major revision. It is composed of several sub-tools targeted at making the development and prototyping of instrumentation applications very simple and efficient. One of its most important components is a compiler for the G programming language.

G is a dataflow language that due to its easy to use and intuitive graphical user interface and programmatic syntax has been very well accepted in the instrumentation industry, especially by scientists and engineers that are familiar with programming concepts but are not professional software developers but rather domain experts. Even though it is very easy to use and very flexible it is build on a very elegant and practical model of computation.

The idea was to provide an intuitive "hardware" view to the programmer, and since most scientists and engineers understood the concept of block diagrams, it became the main syntactical element in LabVIEW™. The semantics are expressed in a structured dataflow manner, which combines constructs from imperative and functional languages. The block diagram consists of virtual instruments or "VI's" (actors) and unidirectional wires (edges) that connect the VI's as shown in Figure 1. VI's are either primitives built in to G or sub-VI's written in the G language.

The user interface is presented through a "front panel" that provides "controls" and "indicators" through which the user sends and receives information, respectively, as shown in Figure 2.

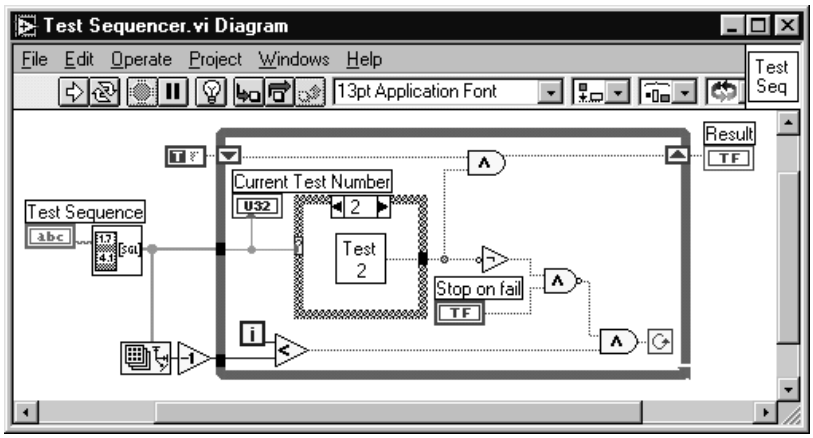

Figure 1 LabVIEW™ Diagram

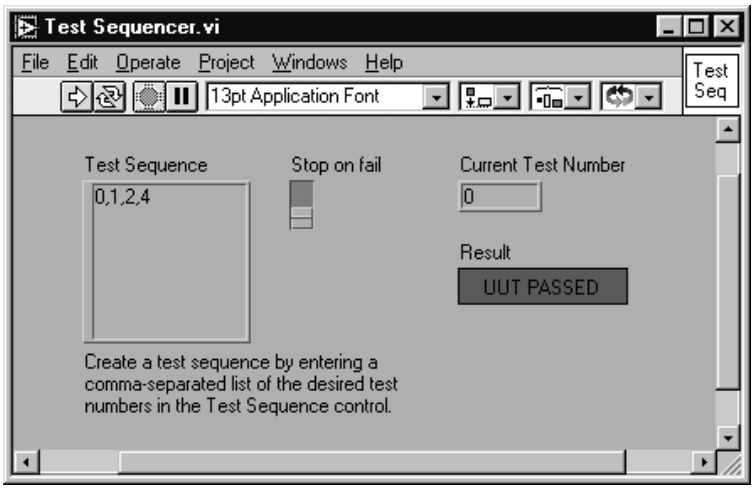

Figure 2 LabVIEW™ Front Panel

3 Motivation

Over the years, LabVIEW™ and G have become dominant in the T&M industry, with many thousands of engineers and scientists using them to develop new applications and libraries that can be used by other developers. In addition to the users, there is a very extensive direct and third party training and support network. In recent years, a new product from National Instruments, BridgeVIEW™, has targeted G as a programming language for the IA industry, and has extended the user and software base.

As the focus turns now to embedded instrumentation systems, it is desirable to be able to re-use that existing infrastructure. The idea is to integrate and adapt to the language as elegantly as possible constructs and paradigm that are used in this new domain, while maintaining backward compatibility with

the existing base. This movement is not unlike others in the industry where an industry standard (e.g. Java) has been enhanced to target more domains (e.g. hardware) to leverage its popularity.

4 Goals of the Project

The main goal of this project is to review the models of computation and technologies in the Ptolemy environment and other methods of software synthesis from dataflow graphs, and to apply them towards the adaptation of G as an embedded software development tool. So far LabVIEW™ has been targeted at powerful PC's, where the structured dataflow, described in Section 6, has been useful to develop high-level instrumentation applications. As we target more specialized processors, distributed systems, real-time systems, and even programmable logic, we need to evaluate the extensibility of G to these domains.

For this literature survey we study two areas in particular: some of the models of computation presented in Ptolemy (SDF, CSSDF, DDF, and PN), and some of the technologies suggested in that environment (multithreading dataflow on Van Neumann architectures, and compile-time scheduling of dynamic constructs in dataflow graphs). Based on the findings from this survey we present a plan for the final paper in Section 9.

5 Review of Some Models of Computation

5.1 Dataflow

A dataflow graph is a directed graph whose *edges* represent data *channels* and whose *vertices* represent *actors* that operate on that data. The number and value of data tokens at the inputs to an actor will determine when the actor will fire. When an actor fires, it consumes some number of tokens on its input channels and produces some number of tokens on its output channels.

Dataflow models can be categorized as synchronous or dynamic, homogeneous or multirate, and multidimensional or unidimensional. Note that there is no hierarchy to these dataflow categories. Although some references list these models in order from simple to complex (HSDF, SDF, DDF, etc.), one can easily create a homogeneous, dynamic, unidimensional model, or a multirate, static, multidimensional model.

5.2 Process Networks

Process networks differ from dataflow in that actors are continuously executing processes rather than single shot functions. Process networks are a superset of dataflow. In process networks (unlike dataflow), any data channel can be read from or written to an infinite number of times. While a DDF graph with

dynamically changing channel thresholds could implement a process network, this would require actors that are specially written to usually consume zero tokens and produce zero tokens except when some internal condition is met. This is somewhat against the philosophy of dataflow, since it implies some communication mechanism other than the data channels and the schedule.

6 Formal Description of LabVIEW™

Before we can extend or improve G using the theoretical techniques being studied today, it is important to characterize G [NI98] using the formal terminology of dataflow languages.

6.1 Categorizing G

The G language is a dataflow language that can be characterized according to the categories given above. Specifically, G is a homogeneous, dynamic, multidimensional dataflow language.

- Homogeneous - G actors produce and consume a single token for each edge in the graph. These tokens can be complex data structures, but they are still single tokens.
- Dynamic - The full G language cannot be statically scheduled. G includes constructs that allow portions of the graph to be conditionally executed based on the input data, so no data-independent static schedule can be created. Like other DDF languages, G has several actors whose firing rules are not dependent on their input data and graphs made of those actors could be statically scheduled.
- Multidimensional - G has full support for multidimensional arrays. The idea of a homogeneous multidimensional dataflow language may seem counter-intuitive, but several VIs in G provide for passing arrays of tokens along wires. Loop constructs in G can be used to combine individual tokens into arrays of tokens, or to separate array elements back into individual tokens. Also, the primitive actors (add, subtract, etc.) can accept arrays of tokens as well as individual scalars.

6.2 Other Properties of G

- Turing Complete: It has been demonstrated that if you can implement a Turing machine in a language, that language is Turing complete.[LP81] A Turing machine has been implemented in LabVIEW™, so G satisfies this condition. G therefore carries some of the baggage of a Turing complete language, such as having unbounded memory size and execution time.

- Bounded communication queues: Although the data structures contained in a token can be arbitrarily large, there can only be one token on any wire at any time. This simplifies the implementation of the communication queues.
- Structured dataflow: Instead of switch, select, and feedback loops, G has programming structures to control program flow. There is a structured case statement that will select one subgraph to execute based on a single input. There are while and for loops in which the user can specify feedback from one iteration to the next. (This is the only feedback allowed in G.) Because G uses programming structures instead of special actors to specify control flow, it is referred to as a structured dataflow language.
- Composability: Because load balancing is not an issue in scheduling homogenous dataflow, G diagrams can be clustered into sub-diagrams without affecting the correctness of the diagram. The only exception is that since G only allows feedback in a loop structure, the partitioning cannot be allowed to create a feedback loop. LabVIEW™ has a command to create sub-VIs from a selection on the diagram. This command automatically detects feedback loops. Furthermore, a node in G can be a VI written entirely in G. On the front panel of a G VI, you specify which user interface elements should be used as inputs and outputs when that VI is used as a sub-VI in a another VI diagram. The sub-VI can be a binary compiled from within LabVIEW™, which allows libraries to be distributed without source. G does not need to know the internal implementation of a sub-VI to be able to schedule it.
- Explicit coupling: G supports non-dataflow communications directly in the diagram. Global variables, local variables, and synchronization primitives can be used to explicitly send data or control scheduling in a VI. This reduces the need to have hidden communication between nodes that might affect the scheduling algorithm.

7 Comparison of G to Other Computational Models

In this project, we hope to apply to G some of the scheduling techniques used by other tools for other dataflow languages. We must therefore compare the G language to these other languages in the hope that similarities in the models will indicate that these techniques will be useful for G. Here we present a summary of some other tools and models that we will be studying.

7.1 Cyclostatic Dataflow

In synchronous dataflow, the firing rules for an actor are fixed from one iteration to the next. This can be limiting in cases where the computation being performed varies periodically. An example is a linear

predictive algorithm in which coefficients are calculated at the beginning of each set of n data points. It would be inefficient to send the same coefficients n times to the actors that will act on the individual data points. In contrast, cyclostatic dataflow allows the firing thresholds to vary periodically.[BEL96] A dataflow programming environment called GRAPE supports cyclostatic dataflow. Work done in this area has shown that cyclostatic dataflow can be statically scheduled by transforming it to an SDF graph.[PPL95] Since the G programming language has loop and case structures instead of general feedback and BDF semantics, it may be possible to identify periodically varying sections of the diagram and schedule them statically. For example, a loop that treats the first element differently than the others could still be statically scheduled even though it contains a case structure.

7.2 Dynamic Dataflow in the Processing Graph Method Tool

The Naval Research Laboratory (NRL) has developed a paradigm called the Processing Graph Method (PGM) and a tool for specifying PGM graphs called the Processing Graph Method Tool (PGMT).[K97],[S97] PGM is essentially a DDF model. Some of the work the NRL has done with PGM involves static analysis of a graph to find segments that should be targeted to different processors.

7.3 Process Networks

In G, VIs do not always execute to completion when their data is available at the inputs. The internal implementation of a VI can choose to wait on some internal condition and return control to the global scheduler so some other VI can get execution time. This is more similar to a process network model than a dataflow model, since the individual VIs have some choice about when to schedule themselves. The actual difference between a dataflow graph and process network is that a process network actor may read or write an infinite amount of data on a particular input or output channel. Since this is not true in G, G is not a process network. However, G's flexible scheduling, hierarchical architecture, and explicit synchronization may allow it to borrow some of the techniques in process networks, such as demand-driven scheduling and the firing of subgraphs. [LP95]

8 Overview of Other Technologies

In addition to studying models of computation as part of our study of the applicability of concepts of software synthesis from dataflow models, we found it useful to study some current technologies in the area of dataflow architectures. Traditionally dataflow has been associated with dedicated machines with hardware architectures that reflected the model of computation very closely. Over the last few years the

research has focused on using mode traditional Van Neumann architectures, taking advantage of the large base of general-purpose computers. This allows us to use much more of the research to complement G. It now becomes important to focus on the coarse grain parallelism, and to schedule possible static portions accordingly. The following subsections describe such research.

8.1 Dataflow Architectures and Multithreading

Lee and Hurson [LH94] present the chronological and architectural migration from pure dataflow machines to coarse grain execution of dataflow programs on general-purpose computers. For these new architectures to be successful the authors are counting on the success of multi-threading, but they warn that the context switch would need to be very lightweight. They also suggest improved hybrid data structures to stream the dataflow. [LH92]

This argument is consistent with the experience in G where instruction level parallelism has not been needed at all times, and optimizations have been made to cluster and statically schedule parts of the diagrams.

This paper hints at the possibility of having a general-purpose processor for the coarse grain parallel execution (software) combined with fine-grain programmable logic elements (hardware) that can execute the fine-grain parallelizable sections. In this circumstance a good hardware-software co-design environment that can efficiently partition and schedule coarse and fine granularity components becomes very important.

8.2 Compile-Time Scheduling of Dynamic Constructs in Dataflow Program Graphs

Ha and Lee [HL95] present the idea of profiling the execution of a system to gather sufficient data for compile-time scheduling of data-dependent dataflow program graphs.

The paper concentrates on how to recognize statically schedulable parts on conditionals, loops and recursion. It would be interesting to apply some of the techniques to gather data on higher level constructs like state machines. It should be noted that G does not have a structured way of describing state machines so they are usually incorporated into conditionals and loop structures. If we can establish a profile for that type of combination of primitives, we could possibly improve schedulability.

9 Conclusions and Projected Work

In this paper we present introduction to the model of computation and accompanying tools provided in LabVIEW™ and the G programming language. We then suggest the adaptation of this language for

embedded systems, based on the work done in Ptolemy and other methods of software synthesis from dataflow graphs.

Based on information reviewed in this paper, we have selected three areas of interest for the implementation portion of the project. In particular, we are planning to do

- An analysis of subsets of G that can be statically scheduled, and an algorithm for generating such schedules
- An analysis of features available in other languages that could be used to extend G
- A proposal for using the G language as a galaxy development in Ptolemy's hierarchical treatment of computational models.

10 References

[BML96] S. S. Bhattacharyya, P. K. Murthy, and E. A. Lee, *Software Synthesis from Dataflow Graphs,* Kluwer Academic Publishers, ISBN 0-7923-9722-3, 1996

[BEL96] G. Bilsen, M. Engels, R. Lauwereins, "Cyclo-Static Dataflow," IEEE Transactions on Signal Processing, vol. 44, no. 2, pp. 397-408, February 1996

[HL95] S. Ha, and E. A. Lee, "Compile-Time Scheduling of Dynamic Constructs in Dataflow Program Graphs," IEEE Transactions on Computers, vol. 46, no. 7, pp. 768-778, July 1997

[K97] D. J. Kaplan, "An Introduction to the Processing Graph Method," Naval Research Laboratory, http://www.ait.nrl.navy.mil/pgmt/Intro_pgm/Final_Intro_pgm.html, 1997

[LP81] H. R. Lewis and C. H. Papadimitriou, *Elements of the Theory of Computation*, Chapter 4, Prentice-Hall, 1981

[LP95] E. A. Lee, and T. M. Parks, "Dataflow Process Networks," Proceedings of the IEEE, May 1995

[LH92] B. Lee, and A. R. Hurson, "A Hybrid Scheme for Processing Data Structures in a Dataflow Environment," IEEE Transactions on Parallel and Distributed Systems, vol. 3, no. 1, pp. 83-96, January 1992

[LH94] B. Lee, and A. R. Hurson, "Dataflow Architectures and Multithreading," Computer, August 1994, pp. 27-39

[NI98] National Instruments, "LabVIEW 5 Software Reference and User Manual", National Instruments, February 1998

[PPL95] T. M. Parks, J. L. Pino, and E. A. Lee, "A Comparison of Synchronous and Cyclo-Static Dataflow," Asilomar Conference on Signals, Systems, and Computers, October 1995

[S97] R. S. Stevens, , "A Processing Graph Method Tool (PGMT)," Naval Research Laboratory, http://www.air.nrl.navy.mil/pgmt, 1997