LabVIEW Based Embedded Design
[First Report]

Sadia Malik
Ram Rajagopal

Department of Electrical and Computer Engineering
University of Texas at Austin
Austin, TX 78712
malik@ece.utexas.edu
ram.rajagopal@ni.com

Abstract

LabVIEW is a graphical programming tool that allows the description of programs using a dataflow based language denominated G. Recently, runtime support for a hard real time environment has become available, making LabVIEW an option for embedded systems prototyping.

Due to its diverse nature, the environment presents itself as an ideal tool for both the design and implementation of embedded software. In this project we study the design and implementation of embedded software, using G as the description language, and the LabVIEW RT real time platform. One of the main advantages of this approach is that the environment leads itself to a very smooth transition from design to implementation, allowing for powerful cosimulation strategies (e.g. hardware in the loop, runtime modeling).

In order to evaluate the effectiveness and possible improvements on G as an embedded software description language, we characterize its semantics and formal model of computation. We also compare G to other models of computation, such as synchronous dataflow, process networks, and finite state machines. We prove that, under certain conditions and semantic restrictions, a non-terminating G program is strictly bounded in memory. We provide a mechanism to always determine a valid G schedule. The theory formalizes the current behavior of the G execution system, and provides insights in how to use G for embedded processing. Also we present an $O(N^3)$ algorithm for detecting non-determinism in a G program.

Finally we propose the development of a state of the art embedded motion control algorithm using LabVIEW as the design, simulation and, implementation tool.

1. Introduction

LabVIEW is a graphical programming environment, developed by National Instruments, based on the dataflow paradigm. It was originally targeted towards the test, measurement, and automation industry.

In recent years there has been a tremendous growth in the embedded software systems market. It was motivated by, among other factors, the reduction in the cost of hardware and the need for fast portable solutions with short time to market. National Instruments developed LabVIEW RT to answer these demands.

The objective of this project is to propose a framework for using the LabVIEW RT software and hardware environment for embedded systems design. Specifically, we propose the design of an embedded motion control system.

To define a consistent framework for embedded design, the underlying model of computation of LabVIEW, the G language, has to be characterized and formalized. In this report we will discuss some of the characteristics of G, compare it to other models of computation and devise a framework to adapt it for embedded systems design and implementation. We will also introduce the embedded motion control problem.

2. Characterizing G

In order to understand G, and consistently use it for embedded software development, we should formalize its semantics and syntax.

A G based program has two components: a diagram and a user interface (front panel) [Figure 1]. Every input and output element in the user interface has its corresponding representation in the diagram, as a source or sink node. The user can change the values of the input at any time during execution, introducing a dynamic element into the diagram.

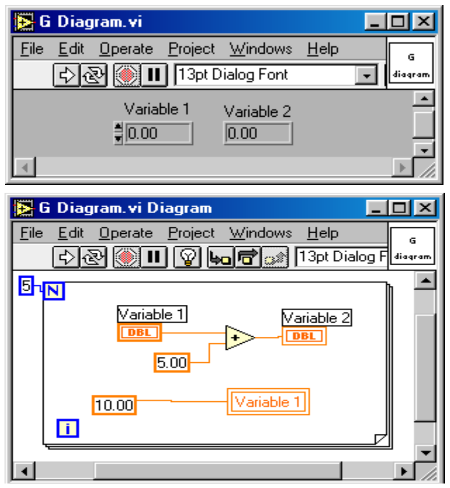

Figure 1: A LabVIEW diagram and front panel

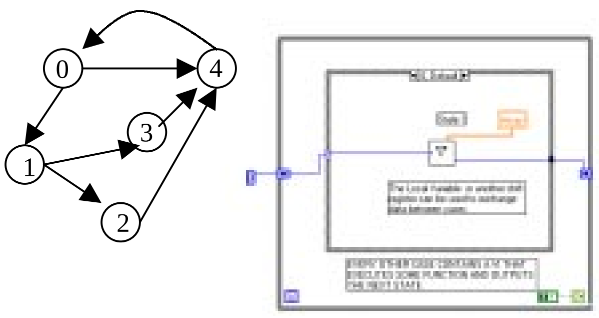

Figure 2: G Finite State Machine

Actors in G are denominated VIs (Virtual Instruments), and can be connected to each other through wires (edges) that allow tokens of multiple dimension to be exchanged (single values, arrays, structures, etc). Global and local variables can be used as means to exchange data (tokens) as well.

2.1 G Properties

The semantics in G are expressed using a combination of constructs from imperative and functional languages, thus constituting a structured dataflow language. A comprehensive characterization of G is presented by Andrade *et al* [AK98]. In this section we will summarize and expand that characterization. The complete G can be characterized as a homogeneous, dynamic, multidimensional structured dataflow based language.

G is homogeneous because every actor consumes or produces a single token for a given edge in the graph. The token can be a very complex structure, whose size can change during execution as long as the dimensionality remains constant.

The complete model of G cannot be statically scheduled [AK98], as G has several structures and actors, whose behavior depends on the input data, such as *case* structures, the *select* VI, and loops controlled by inputs from the user interface.

G is also Turing complete [AK98,K97] and local and global variables allow non-determinism [P95] to arise in G (section 3.1.1).

A very important property is that G is always composable, thus several VIs in a diagram can be gathered into a single one, as long as there are no feedback loops, without affecting the behavior of the program. This can be done because the graph is homogeneous and delayless feedback loops are not allowed in G. Another consequence is that every G graph is directed and has a multiple source and sink structure.

2.2 G and Other Models of Computation

Even though G does not fit exactly into any of the presented models, it shares characteristics with several of them:

Process Networks: By definition a Process Network (PN) is a very generic model of computation, where concurrent processes communicate through unidirectional FIFO channels [LP95]. In G, processes (actors) can also communicate through global and local variables, which are not FIFO, and cannot write an infinite amount of data on inputs/outputs [AK98]. Therefore, the complete G model cannot be classified as a PN. But G keeps some resemblance to PN, as every VI is a small process that generates tokens of arbitrary size (but fixed dimension).

Integer Dataflow (IDF): The homogeneous IDF model [B94] is a model that resembles G, when restricting the use of global and local variables. The main differences are that in G tokens can be multidimensional and flow control is done through *case* structures and *while* loops. A complete G graph can be quasi-statically scheduled [AK98].

Synchronous Dataflow (SDF): A restricted version of G, where *switch* actors, *case* structures, *while* loops, global and local variables, as well as data dependent *for* loops are not allowed, can be modeled as a homogeneous SDF [LM87] and thus statically scheduled. Multidimensional SDF (MSDF) [M96] is an extension to SDF, where actors produce and consume n-dimensional rectangles of data. G cannot be well characterized by MSDF because any array data exchange is done through single multidimensional tokens.

Finite State Machines (FSM): G can be used to express Finite State Machines [LL98]. A standard FSM template for LabVIEW is presented in Figure 2. Note that it is possible to integrate the FSM concept into a dataflow framework. Local and global variables could be used to share data between such state machine and a normal dataflow program.

3. G in Embedded Design

There are many different definitions for embedded software, but an accepted one is a software system, with extremely restricted user interface, that acts on infinite streams of data. The main desired requirements for specifying and executing an embedded program can be listed as [P95]:

1) The program specification should preferably be determinate, and therefore the outputs should be consistent to the inputs, regardless of execution details. Also the program specification should be sample rate consistent and causal [R1].

2) The scheduler should implement a complete execution of the G program, so if the program is non-terminating, it should not deadlock [R2].

3) The scheduler should, if possible, execute a bounded G program in bounded memory [R3].

These requirements are quite natural and express that a well-behaved program should be able to operate without hurdles in standard embedded environments. In this section we will present algorithms and restrictions that guarantee that the main requirements are met, thus allowing G to be transparently used in embedded software design.

In order to derive our algorithms and restrictions, we will assume that for every execution of the G program the front panel input values are fixed. This assumption is reasonable under the hypothesis that the program is an embedded program. Due to space constraints all proofs are presented in the appendix.

3.1.1 Determinism and Consistency

A G graph is always sample rate consistent because it is homogeneous. Also causality in G is guaranteed, because the semantics do not allow delayless feedback loops.

Non-determinism only arises in G when local or global (storage) variables are used as part of a diagram. Because of the fact that G allows multiple reads and writes to a single variable, race conditions may arise. A typical situation is presented in Figure 1, where the execution order of the actors determinates the value of the output *variable 2*.

An efficient algorithm to identify non-determinate programs in G is given in Table 1. This algorithm is based on propositions 1 and 2 below. A flattened diagram is a G diagram where all higher level actors are decomposed into G's basic actors).

Proposition 1: If all actors in a flattened G diagram only read from storage variables, then the program is determinate.

Proposition 2: If an actor A writes to a storage variable, and an actor B reads or writes to the same storage variable, then this program can be determinate if and only if there is a directed path from actor A to actor B or vice versa.

The procedure for creating virtual edges for G structures is presented in the appendix. The "Find Non Determinism" algorithm is of order O(|actors|^3) [CL90].

Find Non Determinism

```
Flatten the G graph.
If (no storage variable write in diagram)
    { program is determinate;}
else
{  Create virtual edges for all G structures;
   Run All pairs Shortest path
        algorithm on graph [CL90];
   For each storage variable S{
      Select a vertice $V_A$ that writes to
           an instance of S;
      For every vertice $V_k$ connected to an instance of S
      {
        If (dist($V_A$,$V_k$) $= \infty$ and dist($V_k$,$V_A$) $= \infty$)
            { Program is non-determinate;
                exit;
            }
      }
}
Program is determinate;
```

Table 1: Algorithm for identifying non-determinate G programs

3.1.2 Bounded Memory Execution

Differently from PN, or even SDF, a scheduler that operates in G does not need to be concerned about memory bounded scheduling. G programs can only be either strictly bounded or unbounded in memory (proposition 3). The main reasons are the homogeneity of the G graph and the syntax restrictions imposed by the language.

Proposition 3: G programs are either strictly bounded or unbounded in memory [M96]. Unbounded memory programs are defined by the use of *build array* actors inside *while* loops or having indexing enabled for *tunnels* in such loops. Therefore a scheduler can always attend R3.

A simple requirement that would force every G program to be strictly bounded in memory is to force every *while* loop to have a maximum count [max_count].

3.1.3 Complete Execution

Because of the semantics of G, and the fact that it is homogeneous, any valid schedule guarantees complete execution of a determinate program. There are no possible deadlocks, unless they are induced by actor behavior (e.g. infinite while loop). Theorem 1 presents a proof that every determinate diagram has a valid execution schedule that attends [R3]. In fact, if the [max_count] requirement is attended, then no G program will ever deadlock due to program specification.

Theorem 1: Given a determinate G diagram, in the sense of section 2.3.1, there is always a valid schedule that consists of a sequential Breadth First Search [CL90] order firing of every actor on the diagram.

4. LabVIEW RT Based Motion Control

Embedded motion control as implemented in the industry today is primarily based on simple algorithms like PID. A majority of manufacturers use the basic PID algorithm with enhancements such as feedforward components and open loop compensation [DTD]. A typical motion control system uses the structure in Figure 3 for its processing and I/O.

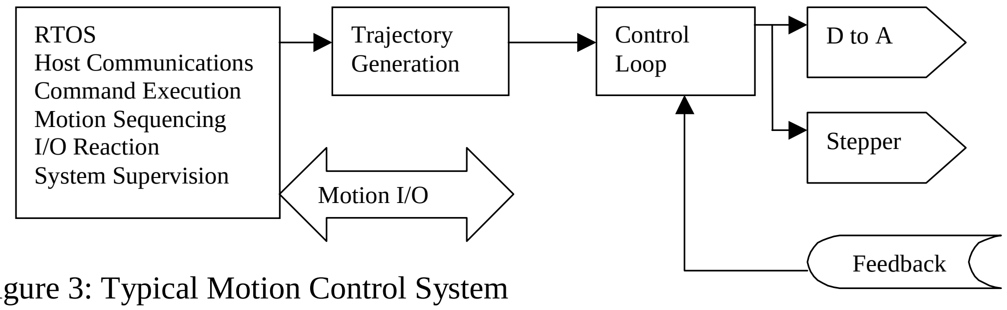

Figure 3: Typical Motion Control System

Commercially available motion control pushes the limits of currently available real time boards and software. Yet, they are inadequate for future applications in areas such as robotics, space applications and other high performance systems. The limitation of existing systems is that their current closed architecture makes it very hard to offer advanced control strategies that require dynamic adaptation capabilities [HL99]. One such example is Multiple input, multiple output state space (MIMO) control . Moreover, there are many situations where users could be interested in defining customized strategies. With current systems this is a hard task, requiring detailed knowledge of the underlying hardware structure.

We propose to design and implement an embedded motion control system that can overcome many of these problems. The essential idea is the seamless integration of the development environment (LabVIEW RT) and hardware (Real Time board). LabVIEW RT can be used to design algorithms, which can be downloaded directly to the board.

The project will require designing a solid framework for integrating the several motion control components. We plan to design our prototype to handle two axis integrated control, using MIMO based strategies. The project requires integrating several software and hardware components as well, posing a very challenging problem.

5. References

[LM87] E. A. Lee and D. G. Messerschmitt, "Static Scheduling of Synchronous Dataflow Programs for Digital Signal Processing", *IEEE Transactions on Computers*, Vol. C-36, No. 2, Feb. 1987.

[CL90] T. H. Cormen, C. E. Leiserson and R. L. Rivest, *Introduction to Algorithms*, McGraw Hill Pub. Co., ISBN 0070131430, 1990.

[B94] J.T. Buck, "Static scheduling and code generation from dynamic dataflow graphs with integer valued control signals", *Proc. of IEEE Asilomar Conference on Signals, Systems and Computers* , Oct. 1994.

[LP95] E. A. Lee and T. M. Parks, "Dataflow process networks", *Proc. of the IEEE*, 83(5):773-799, May 1995.

[P95] T. M. Parks, "Bounded Scheduling of Process Networks", Technical Report UCB/ERL-95-105, University of California at Berkeley, Dec. 1995.

[M96] P. K. Murthy, "Scheduling Techniques for Synchronous and Multidimensional Synchronous Dataflow", Technical Report UCB/ERL M96/79, University of California at Berkeley, Dec. 1996.

[K97] Donald E. Knuth, *The Art of Computer Programming: Fundamental Algorithms*, Addison-Wesley Pub. Co., ISBN 0201896834, 1997.

[LL98] B. Lee and E. A. Lee, "Hierarchical Concurrent Finite State Machines in Ptolemy", *Proc. Int. Conf. on Application of Concurrency to System Design*, Fukushima, Japan, Mar. 1998.

[AK98] H. Andrade and S. Kovner, "Software Synthesis from Dataflow Models for Embedded Software Design in the G Programming Language and the LabVIEW Development Environment", *Proc. IEEE Asilomar Conference on Signals, Systems, and Computers*, Nov. 1998, pp. 1705-1710.

[HL99] S. Han, M. Lee, and R.R Mohler, "Real-time implementation of a robust adaptive controller for a robotic manipulator based on digital signal processors", *IEEE Transactions on Systems, Man and Cybernetics, Part A*, Vol. 29 2, March 1999, pp 194-404.

[DTD] *PMAC2 Reference Manual*, Delta Tau Data Systems Inc.