Presented at the Thirty-second Annual Asilomar Conference on Signals, Systems, and Computers. Pacific Grove, California, U.S.A., November 1998

Software Synthesis from Dataflow Models for G and LabVIEW™

Hugo A. Andrade
Scott Kovner

{andrade,kovner}@natinst.com

National Instruments Corporation

Abstract

The "G" language (in 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 and LabVIEW™ for embedded software development.

G is a homogeneous, multidimensional, dynamic dataflow language. G uses "structured dataflow" semantics to specify high level concepts (e.g. loops). We examine G in the context of other models of computation, such as cyclostatic and dynamic dataflow, and process networks.

G has useful subsets that can be statically or quasi-statically scheduled. In some diagrams, cyclostatic analysis can be used. Parallelism can be further exploited by allowing overlapping execution of loops, and adding array auto-subsetting. Another useful addition would be execution relative to a global clock. Finally, a view manager could present a G program using a different model of computation.

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 intuitive graphical representation and programmatic syntax has been 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. Though it is easy to use and flexible, it is built on an elegant and practical model of computation.

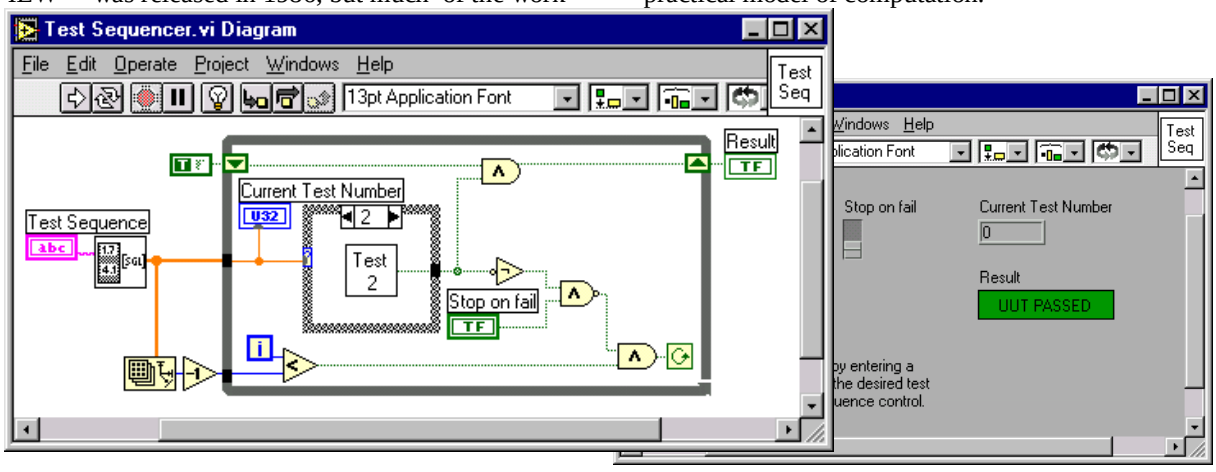

Figure 1 LabVIEW™ Diagram and Front Panel

The idea behind G was to provide an intuitive "hardware" view to the programmer. Since most scientists and engineers understood the concept of a block diagram, it became the primary 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 on the left in Figure 1. VI's are either primitives built into 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 on the right in Figure 1.

3 Motivation

Over the years, LabVIEW™ and G have been accepted well 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 paradigms 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 this research

The main goal of this research is to review the models of computation and technologies in the latest work done on 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 desktop 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.

In this discussion, we provide a formal description of the G programming language (Section 5). We identify subsets of G that can be statically scheduled (Section 6). We present features available in other languages that could be used to extend G (Section 7). Finally, we discuss the combination and integration of G with other models of computational (Section 8).

5 Formal description of LabVIEW™

Before we can extend or improve G using the theoretical techniques being studied today, it is important to characterize G [8] using the formal terminology of dataflow languages [1, 9].

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 (DF) models can be categorized as synchronous (S) or dynamic (D), homogeneous (H) or multirate, and multidimensional (M) or unidimensional. Note that there is no hierarchy to these dataflow categories.

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.

5.1 Categorizing G

G is a homogeneous, dynamic, multidimensional dataflow language:

- Homogeneous - G actors produce and consume a single token for each edge in the graph.
- Dynamic - G includes constructs that allow portions of the graph to be conditionally executed based on the input data.
- Multidimensional - G has full support for multidimensional arrays. Loop constructs in G can be used to combine individual tokens into arrays of tokens, or to separate array elements back into individual tokens. This is known as "auto-indexing".

5.2 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[4]. A Turing machine has been implemented in G, so G satisfies this condition.
- 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.

- 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. (No other feedback allowed in G.)
- 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. Furthermore, a node in G can be a VI written entirely in G. 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 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.

6 Static scheduling in G

Current run-time implementations of G employ a dynamic scheduler to control the firing of each VI in a diagram. Embedded systems often do not require such scheduling, and cannot afford the overhead of such a scheduler in the run-time environment. In order to target G programs to embedded processor code or a hardware description language, it is necessary to find subsets of G that can be statically scheduled [1, 6, 7].

6.1 Synchronous VI's and loops

Recall that the actors on a dataflow graph are atomic, while the actors in a process network represent self scheduled processes that communicate via the queues (arcs) between the actors [5]. A program written in the G language describes a simple dataflow process network, rather than just a dynamic dataflow program. Specifically, VI's in G complete either synchronously or asynchronously. VI's that complete asynchronously cannot be scheduled statically since the static scheduler cannot know when the VI will complete. However, the synchronous VI's resemble synchronous dataflow (SDF) actors (in Ptolemy [9]), and can be statically scheduled.

A G compiler should analyze loop constructs to determine if they can be statically scheduled. A "for" loop with a constant loop count can be statically scheduled. On the other hand, "for" loops with data-dependent loop counts and "while" loops that have a data dependent termination condition can not be statically scheduled. This becomes clear if you notice that a loop with a count of either zero or one is equivalent to an if-then statement, which can be used within another loop in G to implement a Turing machine, which cannot be statically scheduled. Others have proposed statistical methods for statically scheduling loops [3].

6.2 Quasi-static scheduling

In traditional binary dataflow (BDF), there are nodes which demultiplex data ("forks") and nodes which multiplex data ("join"). The fork and the join can be placed anywhere in the diagram. It is the responsibility of the BDF scheduler to find finite complete cycles in the graph so that a quasi-static schedule may be generated. Unfortunately, since BDF is Turing-complete, the problem of finding finite complete cycles is undecidable [1].

G, on the other hand, uses a "case" structure instead of "fork" and "join" nodes. A case structure describes alternate VI frames that must all produce the same number and type of tokens at the output of the structure. With such a structure, no analysis is necessary to find a quasi-static schedule; each frame of VI's can be statically scheduled and represents one data dependent alternative in the quasi-static schedule. Therefore, G case structures provide the power of boolean decisions in the diagram without introducing the scheduler complexity of BDF.

6.3 Cyclostatic dataflow

In a G diagram, case structures are often used inside of loop structures. Sometimes, the iteration count is used as the condition for the case structure. In those instances, the loop and case structures represent a cyclical firing of the frames inside the case structure. This corresponds to a cyclostatic dataflow (CSDF) program. Such a diagram can be statically scheduled by transforming it into a cyclostatic graph and applying the appropriate scheduling algorithms for cyclostatic dataflow [2, 10].

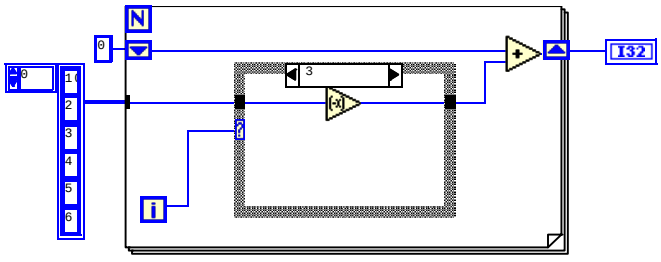

Figure 2a Cyclo-static G Diagram

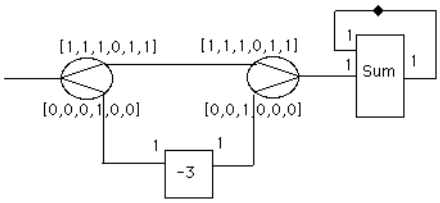

Figure 2b CSDF Representation

6.4 Scheduling algorithm

The homogeneity and structured programming constructs in G make the creation of a static schedule relatively simple, as iterations within a periodic schedule are not an issue [1]. The main issues are the rearranging of code dependencies (to satisfy additional constraints such as multi-rate loops serialization and/or parallelization) and code generation (to have a good tradeoff between memory size and performance).

7 Extending G

In this section we present three extensions to the G language based the concepts found in the environments studied: multi-dimensional dataflow, timed dataflow, and overlapping execution.

7.1 Multidimensional dataflow

G tokens can be multi-dimensional arrays of any data type. Normally, a VI must be specifically written to accept these arrays. G provides extra support for passing arrays into loop structures. The array can be auto-indexed, which means that the i^{th} element of the array is automatically passed into the i^{th} iteration of the loop. Furthermore, a loop with an unspecified loop count will automatically run for a number of iterations equal to the size of the smallest auto-indexed array.

This feature could be extended to provide more of the power of multidimensional dataflow. For example, instead of simply auto-indexing the array, a loop could be made to auto-subset the array. In other words, an array of n elements should be able to pass into each iteration an array of m elements, and the loop should execute n/m times. This would accommodate passing large arrays to VI's that are written to handle specific smaller sizes of data. For example, a two dimensional FFT could be implemented using a VI that operates on 64 pixel by 64 pixel regions of the data. This feature becomes even more useful if the iterations are allowed to be executed in parallel, as is suggested in Section 7.3 "Overlapping Execution".

7.2 Timed dataflow

The concept of having time information is not new to G. It is available today in the form of timers and delays, mainly in loops. It is also available as a configuration option for many I/O libraries that do hardware based timing. Here we would like to introduce the concept of time at the firing level for each VI. This will be very useful in the overlapping execution described in the next sub-section.

We will do this by allowing tags on source VI's that describe start times or periods. This will allow the scheduler and not the VI to control the notion of time in the system. Using a global clock we can precondition the triggering of a given VI. If know the duration of the VI, or we can put bounds on it, we can generate a pessimistic "pseudo-static" schedule, because we know that we will be able to iterate on a timely manner. Through the use of exceptions (error clusters in G) we can deal with the case where a VI does not meet the schedule. We can also deal with multi-rate flows by incorporate sub-structures, and using the multi-dimensional characteristics described above to accumulate and pass the multiple data sets.

7.3 Overlapping execution

The current G scheduler creates an artificial sequential dependence between independent iterations of a loop. As was seen in section 7.1, unrolling loops can help us achieve multi-dimensional dataflow. In this case it will allow us to achieve greater parallelism.

Overlapping execution need not be limited to loops. We can allow multiple firings of the same VI under any circumstance. Notice that we can still maintain the single element queue paradigm currently provided in G by making the schedule demand driven. This would allow for a very intuitive description of pipelined execution, which can be mapped to highly parallel execution environments, like FPGA's. In the case of timed executions where we know the timer period is larger than the execution time, we can build a static schedule and guarantee this demand driven overlapping execution, given that the dataflow allows it.

We can also provide queues that are larger than one element. In un-timed diagrams, this would allow us to compensate for different rates of arrival for firing events. In this case, the scheduling would be data driven, and overflow at the queues is possible. A variation where the arrival of events is cyclical we can apply a cyclic schedule that would allow us to predict the number of queue elements required.

8 G and other models of computation

Even though providing a simple and consistent single model of computation is key to G, it is necessary to integrate G with other environments. LabVIEW™/G currently supports DLL calls and remote control in an imperative environment. We show how LabVIEW™/G can be integrated with other dataflow environments.

There are two approaches to integration. The first is by actually switching models of computation, and providing interfaces to them. The second one is to provide an alternate view for the same model.

In the first approach, we integrate with Ptolemy, for example, by following the wormhole paradigm and interface they have provided. Ptolemy describes in great detail the interaction between different models of computation. G, with its original un-timed semantics, can be integrated in a similar manner to other dataflow environments, and other domains can be integrated with it in a similar way that they were integrated with other dataflow models of computation in Ptolemy. The main issue is that each VI can execute indefinitely once fired, so that integration with timed domain would be more complicated. The concept of timed events discussed in section 8.1 is useful in providing a time-reference for such integration. Untimed dataflow integration is trivial.

A second approach is to provide a view manager that allows developers to visualize a segment of G code in a domain that is more intuitive to them. An interesting case here is the FSM view. G can already implement FSM well through the use of a 'case' statement with an outer loop. Parallel state machines are depicted as parallel loops and interaction through globals, if needed. We can build a view manager that provides a more intuitive view of the FSM, i.e. a traditional Mealy or Moore diagram, with bubbles, arc, and events. The manager would then create and manage a FSM template instance. The user would be able to view and modify actions, and possibly the state variable assignment, but would allow the control to be managed.

9 Conclusion

The G programming language is directly applicable to embedded system development as is. We have proposed several extensions:

- Static Scheduling: In the existing form of the language, there are useful subsets that can be statically scheduled. In particular, synchronous VI's and certain types of loops can be statically scheduled. Quasi-static schedules can be derived from the case structures, and cyclostatic analysis can be used on certain combinations of loops and case structures.
- Exploiting Parallelism: As a dataflow language, G already describes the parallelism in a program. This can be further exploited with overlapping execution of loops, and by augmenting the auto-indexing feature with array auto-subsetting.
- Timed VI's: It would be useful in a G program to describe when or how often a VI should execute relative to some global clock. In current mechanisms for timing in G, the scheduler cannot use the timing information. By tagging source VI's with start information, we can allow the scheduler to generate a real-time periodic schedule. We can also verify that a VI has completed execution by a user specified deadline. Furthermore, multi-rate diagrams can be represented by a hierarchical structure that communicates using arrays of data.
- Multiple equivalent views for the same model.

G and LabVIEW™ already provide a productive environment for the development of dataflow programs. Some of the enhancements proposed here would complement it well, especially in the area of embedded systems, where small, fast, and determinate executions are key. For implementation of such systems, the homogenous, dynamic, timed, and parallelizable model we propose for G is very useful.

10 References

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

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

[3] 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, Jul. 1997.

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

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

[6] 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, Jan. 1992.

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

[8] National Instruments, *LabVIEW™ 5 Software Reference and User Manual*, National Instruments, Feb. 1998.

[9] Ptolemy Project, *The Almagest: A Manual for Ptolemy*, Ptolemy Project (http://ptolemy.eecs.berkeley.edu/papers/almagest/index.html), 1997.

[10] 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*, Oct. 1995.