Hindawi Publishing Corporation
EURASIP Journal on Embedded Systems
Volume 2008, Article ID 162747, 14 pages
doi:10.1155/2008/162747

Research Article

A Rapid Prototyping Tool for Embedded, Real-Time Hierarchical Control Systems

Ram Rajagopal,[1] Subramanian Ramamoorthy,[2] Lothar Wenzel,[3] and Hugo Andrade[3]

[1] *Electrical Engineering and Computer Sciences, University of California at Berkeley, Berkeley, CA 94720-1770, USA*
[2] *School of Informatics, The University of Edinburgh, Edinburgh, EH9 3JZ, UK*
[3] *National Instruments Corp., Austin, TX 78759, USA*

Correspondence should be addressed to Ram Rajagopal, ramr@eecs.berkeley.edu

Received 24 May 2007; Revised 25 February 2008; Accepted 17 May 2008

Recommended by Shuvra Bhattacharyya

Laboratory Virtual Instrumentation and Engineering Workbench (LabVIEW) is a graphical programming tool based on the dataflow language **G**. Recently, runtime support for a hard real-time environment has become available for LabVIEW, which makes it an option for embedded systems prototyping. Due to its characteristics, the environment presents itself as an ideal tool for both the design and implementation of embedded software. In this paper, we study the design and implementation of embedded software by using **G** as the specification 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). We characterize the semantics and formal model of computation of **G**. We compare it to other models of computation and develop design rules and algorithms to propose sound embedded design in the language. We investigate the specification and mapping of hierarchical control systems in LabVIEW and **G**. Finally, we describe the development of a state-of-the-art embedded motion control system using LabVIEW as the specification, simulation and implementation tool, using the proposed design principles. The solution is state-of-the-art in terms of flexibility and control performance.

1. INTRODUCTION

LabVIEW (Laboratory Virtual Instrumentation and Engineering Workbench) is a graphical programming environment, developed by National Instruments Corp., based on the dataflow paradigm. It was originally targeted towards the test, measurement, and automation industries.

In recent years there has been tremendous growth in the embedded software and systems market. It is driven by the need for low cost, fast and portable solutions with short time to market. National Instruments Corp. developed LabVIEW Real Time (LabVIEW RT) to cater to these demands. The LabVIEW RT environment includes the original LabVIEW environment, as well as an ETS/vxWorks-based real-time hardware system that includes modules for data acquisition and control.

In the LabVIEW environment, programs are described using the **G** programming language. Broadly, this language can be understood as a structured dataflow programming language. The environment integrates a compiler and scheduler for the **G** language. In LabVIEW RT, these compiler and scheduler have been extended to generate executable programs from **G** that can be executed in real-time operating systems, such as RTOS. Using the current compiler and scheduler, the complete language specification can be executed in the system, but only with soft real-time constraints.

The objective of this paper is to outline a framework for using the LabVIEW RT software and hardware environments for embedded systems design in a principled way, avoiding issues that compromise real-time operation performance. Whenever required, we consider specifically LabVIEW RT version 7.1. We summarize our contributions.

In Section 2, we provide a formal description of the **G** language. We determine some important characteristics of the language. We compare the **G** model of computation to other comparable models in the literature.

In Section 3, we show how programs can be specified in **G** in order to satisfy some important requirements

for programs that define embedded systems. We develop principles so that **G** programs compiled and scheduled by LabVIEW RT are sound according to the stated requirements. We also compare the LabVIEW RT environment and some other tools available for embedded systems design and deployment.

In Section 4, we show how a complete solution for an embedded control system can be developed and deployed using the proposed environment. Our chosen application is embedded motion control, a general task that is a cornerstone of many manufacturing automation applications. It is also a staple hierarchical control system. The algorithms used to implement the solution use the principles developed in previous sections. We propose some novel approaches which are enabled by the environment, among them an online design based on qualitative control definition. The design is performed online, as the system is operating.

In Section 5, we discuss the changes in standard control design practice that become possible by using LabVIEW RT and **G**. Such changes should also be available to any design tool that implements the toolset currently available in the proposed language and system. In Section 5, we conclude and propose future works.

2. THE G LANGUAGE

In this section, we explore the specification and properties of the **G** programming language. We also compare the language to other dataflow specification languages in the literature.

2.1. Specification of G

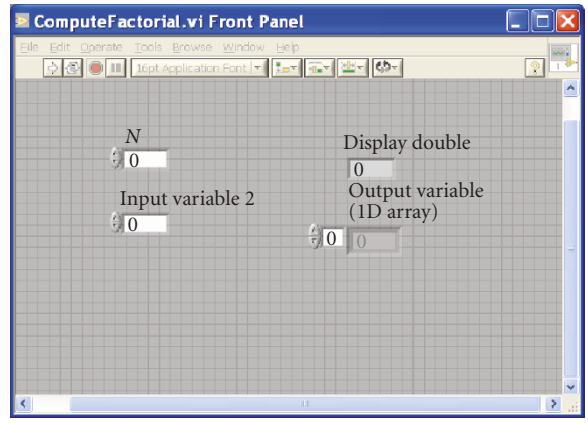
A **G** program has two components [1]: a diagram and a user interface (called *front panel*) as shown in Figure 1. Every input and output element in the user interface has its corresponding representation in the diagram, as a source or sink node, respectively. The user can change the values of the input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

An actor in **G** is called a Virtual Instrument (VI). VIs can be connected to each other through wires (edges) that allow tokens to be exchanged. Each edge can only contain a single data value at time. On the other hand, tokens can have multiple dimensions such as single real numbers or strings, n-dimensional matrices of numbers and structures that contain numbers and strings.

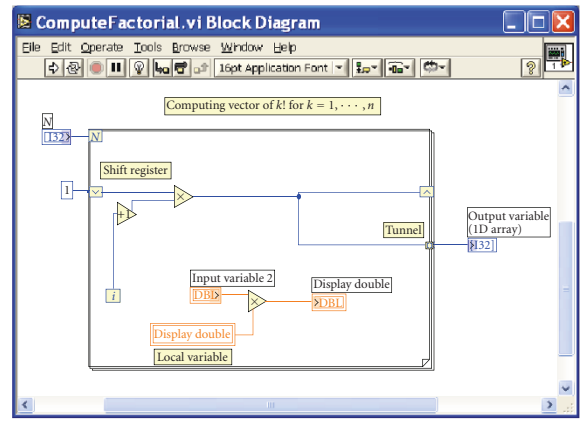
VIs have multiple inputs and multiple outputs. Each input has an incoming wire and each output has an outgoing wire. So for example, if an actor has three vector outputs that are input to another actor, three wires have to be used, since each wire can only hold a single value. VIs fire when all the inputs to them are available, in standard dataflow fashion. Many of the basic actors (VIs) provided in **G** are also polymorphic, as a convenience to developers.

Some basic data types in **G** are booleans, integers, reals, strings, one-dimensional arrays, multidimensional arrays, and structures that contain different types, called bundles.

Typical actors are standard arithmetic operators, standard string operators, and vector and array manipulation



(a)



(b)

Figure 1: (a) Front panel and (b) diagram of a **G** program to compute factorials and a sequential product in the LabVIEW environment. Shown in the diagram is a for loop structure.

operators. An important operation is the build array construct. It takes multiple arrays as an input, and outputs a single array that contains all the elements of the input arrays. Notice that multiple input tokens are consumed and only a single output token is produced, but the produced output token uses at least as much memory as the total memory used by input tokens. This is one of the three methods for growing tokens. The others are explained below.

G allows the specification of global and local variables that can be used as a means to exchange data. Global and local variables, *front panel* inputs, and outputs appear as actors in the **G** diagram. Each input control, output indicator, global or local variable can be represented as more than one node in the diagram, with an attached read or write property. The notion of input control and output indicators has more to do with the original historical intended meaning of these constructs.

There are four main imperative structures in **G**: the case structure, the for loop, the while loop, and the sequence structure. Other structures, such as timed loops and feedback node, are available in current versions of

LabVIEW RT. We do not address them here, but their semantics satisfy the properties discussed in the paper.

All structures are integrated into the general programming paradigm by working as a capsule that contains all the actors inside it. The structure executes when all inputs to the actors inside it become available. Tokens enter and exit the structures through tunnels. Graphically, the structure is appropriately represented as an object that envelops the selected actors (Figure 1(b)).

The for loop is a standard loop where at each iteration all the VIs inside it are fired as data to their inputs and outputs become available. The next iteration is only executed after all VIs in the loop have fired. The number of iterations is specified as an input to the loop actor.

In the situation in Figure 1(b), for example, the product between variable `Display Double` and `Input Variable 2` is computed and stored in `Display Double` through a local variable. In parallel, the current loop index i is multiplied by the last factorial variable calculation stored in a shift register. The output is put back in the shift register. Both of these parallel operations are only executed again when the loop executes again.

The for loop includes two loop specific features: the shift register and a tunnel. The shift register is a construct that stores the token input to it (right-hand side in Figure 1(b)) and makes it available at the next iteration (left-hand side in Figure 1(b)). It is a memory. A shift register can be extended to remember any number of values, but this number is fixed prior to the compilation of the program. The output tunnel construct is also a form of local memory. It can be configured to either remember the last value input into it, or more interestingly to grow a multidimensional token by appending incoming tokens. In the example discussed, the output token from the tunnel will be an array of integers of size N. The behavior of the tunnel can be reproduced by using a shift register fed with the output of a build array, which has as inputs the output of the shift register and the token connected to the tunnel.

The while loop is similar to the for loop except that a conditional statement inside the loop determines when the loop stops executing. The conditional statement can be a boolean output of an actor built out of comparison functions available in **G**.

The sequence structure is a structure comprised of multiple frames. Actors inside each frame can receive inputs from outside the structure, as well as from VIs in previous frames. The sequence structure enforces an ordered execution of the actors firing, having priority over the token driven firing.

Finally, the case structure is a structure comprised of multiple frames with a selector input. According to the token that arrived in the selector input, only one frame is executed. The only caveat for the case structure is that all frames have to produce the same outputs to the remaining program outside the structure. This means that the frame selection can only change the value of the token, but not their type nor the number of tokens outputted.

Feedback loops are not allowed in **G**, so no actor can be part of a feedback loop without a delay. Even if the output of a local variable is connected to the input of a local variable, the data firing rule imposes an implicit ordering.

2.2. Properties of G

A comprehensive characterization of **G** is presented in [2] using formal terminology of dataflow languages [3]. In this section, we summarize and expand on that. **G** can be characterized as a homogeneous, dynamic, multidimensional structured dataflow based language.

G can be categorized as a structured dataflow language since the semantics are expressed using a combination of constructs from imperative and functional languages.

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

G is multidimensional since tokens can have multiple dimensions and dynamic since there are constructs that allow part of the program graph to be conditionally executed based on data.

The complete model of **G** cannot be statically scheduled because it has several structures and actors whose behavior depends on input data. Examples of this include case structures and loops controlled by inputs from the user interface.

G is Turing complete [2, 4] and local/global variables allow nondeterminism [5] to arise. One example of nondeterminism is shown in Figure 1(b), where the value of `Display Double` depends on `Input Variable 2`. If the value of `Input Variable 2` is changed in the user interface while the loop executes, the output value of `Display Double` will depend on whether the change was read in the current or the next iteration. This point is elaborated in Section 3.

A very important property is that **G** is always composable. 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 delay-less feedback loops are not allowed. Another consequence is that every **G** graph is directed and has a multiple source, multiple sink structure.

2.3. G and other models of computation

Although **G** does not fit exactly into any of the presented models, it shares characteristics with several of them as described below.

(i) Process Networks (PN): By definition a Process Network (PN) is a very generic model of computation, where concurrent processes communicate through unidirectional FIFO channels [6]. In **G**, processes (actors) can also communicate through global and local variables. These variables are not FIFO and cannot write an infinite amount of data on inputs/outputs [2]. 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 with fixed dimension.

(ii) Integer Dataflow (IDF): The homogeneous IDF model [7] is a model that resembles **G** by 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 [2].

(iii) Synchronous Dataflow (SDF): A restricted version of **G**, in which switch actors, case structures, while loops, global and local variables, and data dependent for loops are not allowed, can be modeled as a homogeneous SDF [8] and thus statically scheduled. Multidimensional SDF (MSDF) [9] 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 in **G** is done through a single multidimensional token. Also another important difference is that every actor in **G** consumes (produces) a single token from each of its inputs (outputs).

(iv) Finite State Machines (FSM): **G** can be used to express Finite State Machines [10]. A standard FSM template can be easily constructed for **G** (e.g., [1]). 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. EMBEDDED DESIGN IN LabVIEW AND G

There are many different definitions for embedded software. A popularly accepted definition is that it is a software system with extremely restricted user interface that acts on infinite streams of data. The desired requirements for specifying and executing an embedded program can be listed as follows [5].

(i) *Requirement 1*. The program specification should preferably be determinate, and therefore the outputs should be consistent with the inputs, regardless of execution details. Also the program specification should be sample rate consistent and causal. In a sample rate consistent program, actors consume and produce tokens in a balanced way, that is, we can find integer firing rates such that the dataflow can be executed repeatedly. In a causal specification, outputs of each actor depend only on current and past inputs.

(ii) *Requirement 2*. The scheduler should implement a complete execution of the program so that a non-terminating program does not deadlock.

(iii) *Requirement 3*. The scheduler should execute a bounded program in bounded memory. A bounded program is a program where the number of tokens at every edge is bounded by some finite constant in a complete execution of the program. In the **G** context, since each edge can only have one token, but tokens can grow in size, a bounded program is a program whose memory requirements are bounded.

These requirements ensure that a well-behaved program operates properly in embedded environments. Notice that Requirement 1 refers directly to the specification language, whereas Requirements 2 and 3 refer to how the compilation environment and the scheduler are able to handle the specification language. In some situations, the specification language is such that the scheduler requirements are enforced directly by constraints in the language itself. For example, in homogeneous synchronous dataflow systems, there is no possibility of deadlocks.

In this section we will present algorithms and programming guidelines that guarantee that the main requirements are met, thus allowing LabVIEW RT and **G** to be transparently used in embedded software design, while loosing minimal expression power. We then proceed to compare LabVIEW-RT and **G** with other standard popular embedded programming languages.

We assume that the standard compiler and scheduler in LabVIEW-RT is used. In order to derive our algorithms and restrictions, we will also assume that for every execution of the **G** program the front panel input values are fixed. In typical industrial situations, the values are indeed fixed, and the program is autonomously run such as in the motion control example in this section.

3.1. Determinism and consistency

A **G** graph is always sample rate consistent because it is homogeneous. Also causality is guaranteed because the semantics do not allow for delay-less feedback loops.

Due to the semantics, non-determinism only arises in **G** when local or global (storage) variables are used as part of a diagram. Due to the fact that **G** allows multiple reads and writes to local and global variables, race conditions may arise. For example, a simple program where two instantiations of a single local variable are connected to different constants is nondeterministic.

A polynomial algorithm to identify nondeterminate programs in **G** is given in Algorithm 1. This algorithm is of complexity $O(A^3)$, where A is the number of elementary actors in the graph, and is based on Claims 1 and 2 given below.

A flattened graph refers to graph representing a VI where all higher-level actors are decomposed into basic **G** actors and embedded in a single diagram. Each actor is a node in the graph. For the case structure and loops, two virtual nodes are added: a virtual input node and a virtual output node. The virtual input node receives connections from all actors whose outputs where incoming into the loop structure. This node has as outputs the same values as the inputs. The outputs are connected to the nodes inside the structure, matching the way they are originally connected in the diagram. Similarly, a virtual output node receives as inputs all the outputs that were coming out of the structure. This node then outputs to nodes representing the actors that received information from the structure. These virtual nodes are just a way to represent the synchronism that a structure imposes. The sequence structure includes one extra synchronization node for each frame. The proposed construction is always possible since **G** does not allow feedback loops without delays.

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

Find non-deterministic diagram:
(a) Create flattened **G** graph corresponding to original diagram;
(b) If (no storage variable write in program) {
program is determinate;
return;
}
(c) Run All Pairs Shortest Path algorithm on graph;
Denote by dist(V_1, V_2) distance between nodes V_1 and V_2;
(d) For each storage variable S in the original diagram {
Determine the corresponding set $\mathcal{V}_S$ of vertices in the graph.
Determine the set $\mathcal{V}_i(S)$ of all vertices V_i such that (V_i, V_S) is an edge in the graph for some $V_S \in \mathcal{V}_S$.
Determine the set $\mathcal{V}_o(S)$ of all vertices V_o such that (V_S, V_o) is an edge in the graph for some $V_S \in \mathcal{V}_S$.
For each node $V_i \in \mathcal{V}_i$ {
For each node $V_o \in \mathcal{V}_o$ {
If (dist$(V_i, V_o) = \infty$ or dist$(V_o, V_i) = \infty$) {
Program is nondeterminate;
return;
}
}
}
(e) Program is determinate;

Algorithm 1: Algorithm for finding non-determinism in a **G** diagram. For creating flattened graph see Section 3.

The proof of the claim is direct since there are no write operations to storage variables, so they are constants. By virtue of this fact, theprogram is determinate.

Claim 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 claim is proved by the following observation: a directed path (A, B) or (B, A) implies that there is a data dependency between A and B (also data dependency implies a directed path, due to **G** syntax). Therefore, in any valid **G** schedule, actor A will be strictly fired, respectively, before or after actor B. So these operations do not create a race condition and the program can be determinate.

The algorithm in Algorithm 1 exploits the fact that nondeterminism in **G** only arises when edge dependencies in the diagram do not prevent race conditions. Based on the observations in this subsection, we can propose a first principle as follows.

Design Principle 1

Multiple copies of the same local or global variable should only be used inside sequence structures and will always reside on different frames. In the case of the global variable this means the sub VIs that use the globals, and are at the same aggregation level (e.g., subVI versus sub-subVI) should be on different frames of a sequence structure.

3.2. Bounded memory programs

Differently from PN and SDF, a scheduler that operates in **G** does not need to be concerned about bounded memory scheduling as long as *Design Principle* 1 is followed, enforcing determinism. **G** programs can only be either strictly bounded or unbounded in memory. This is due to homogeneity of the **G** graph and the syntactical restrictions imposed by the language.

Claim 3. **G** programs are either strictly bounded or unbounded in memory [9].

Unbounded memory **G** programs are defined by the use of build array actors inside while loops or having indexing enabled for tunnels in such loops. 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.

In **G**, there is no possibility of token accumulation, due to the data driven paradigm and homogeneity (the balancing equations of SDF are automatically satisfied [3]). However, token sizes can increase, thus creating the possibility of unbounded memory usage. Using the build array VI generates increased token sizes. The only way unbounded memory usage could be achieved is by having an infinite stream of linked build array actors. Finally, this is only possible if such actors are used inside while loops that run indefinitely.

Also enabling "indexing" in output tunnels of a VI generates tokens with increasing size. Indexed output tunnels

are just a graphical representation for a build array combined with a shift register structure.

Real-time performance degrades in programs where memory usage is increasing or unpredictable, making it attractive to keep **G** programs bounded in memory. We can state the following.

Design Principle 2

While loops should always have a maximum iteration condition. Furthermore, for efficiency, whenever possible, arrays should be preinitialized using initialize array and filled up using replace array subset actors, avoiding the memory uncertainties of the build array actor.

3.3. Complete execution

Based on the semantics of **G**, and because it is homogeneous, it is clear that any valid schedule guarantees complete execution of a determinate program. There will be no deadlocks unless they are induced by actor behavior (e.g., infinite while loop). Notice that Requirement 2 is again a requirement in the scheduler in LabVIEW, but that due to the structure of **G**, using *Design Principle* 1 and *Design Principle* 2 we are able to guarantee sound behavior for the system independently of scheduling decisions. Thus we have the following.

Claim 4. Given a determinate **G** diagram, there is always a valid schedule that consists of a sequential Breadth First Search [11] order firing of every actor on the diagram. This implies that the **G** diagram can never deadlock. Any scheduler for **G** will be able to satisfy Requirement 2.

Claim 4 states that every determinate diagram has a valid execution schedule that satisfies Requirement 3. In fact, if the maximum count requirement is satisfied, then no **G** program will ever deadlock due to program specification.

To prove the claim true, using the composability principle, we can construct a diagram where every structure (while loops, for loops, case and sequence structures) is inside a separate SubVI. We also add a virtual source that is connected to every other source in the graph. In this diagram the valid schedule consists of firing all actors in a breadth first search (BFS) order, starting at the virtual source. The BFS search order guarantees that all actors (including switches) will be appropriately fired, as all inputs to every actor will already be available when it is its turn to be executed. The subVIs that include case structures can also be fired on the BFS order, except that a specific path will be chosen depending on the control input. But still a path should exist due to **G** syntax restrictions. The diagram of composed subVIs that contain while loops or for loops can be looped and fired repeatedly. In fact if we are dealing with a **G** program with maximum loop counts, then all loops can be replaced by their decomposed versions. Now we have the original situation again and thus the BFS search can be applied.

3.4. LabVIEW-RT/G and other tools

In many applications it may be required that a certain periodic schedule be executed within a hard-timed loop. For example, in a digital control application hard timing is extremely important to ensure system stability [12].

LabVIEW-RT was developed to allow a hard timed execution of a **G** diagram. It consists of an execution kernel (RT Engine) that runs the **G** diagram on the RTOS real-time operating system. The execution kernel is supported by a standard industrial PC (based on a PXI chassis) equipped with a National Instruments data acquisition board. In this work, development was done on a host desktop computer using a LabVIEW-RT interface. The host computer is linked to the RT embedded controller system, to download the program and update the user interface.

We can compare LabVIEW and **G** to some popular embedded system development languages and tools. We focus on hybrid languages [13], since they seem to be the most adopted in current real-time system development.

One feature of LabVIEW and **G** is that the real-time runtime environment, and the simulation environment use the same compiler and scheduler, which facilitates embedded control systems development, since usually in this domain extensive simulations are performed, including hardware in the loop tests, before a first prototype.

(i) Esterel [14] is a hybrid language that combines constructs of imperative languages with some facilities of data flow languages, such as concurrency and preemption. The language has a synchronous model of time. Signals can be present or absent, and in each clock cycle the program awakens reads inputs and produces outputs. The language is determinate in that signal checks are always performed before block execution. Compared to Esterel, LabVIEW and **G** offer two main differences: signals are always present and programs are represented as diagrams. Furthermore, causality violations are easily avoided in **G** if proposed design principles are followed.

(ii) SDL is a graphical language defined by the ITU [15]. A program consists of concurrent FSMs, each with a single input queue and communication channels to communicate among them. Execution might be nondeterministic since the order of arrival of tokens to the different queues depends on execution speed of each FSM. An SDL program can be emulated in **G** using the concurrency available in the language, but this sacrifices determinism. A better approach is to design a sequenced parallel FSM implementation using the sequence structure. The main advantages of LabVIEW and **G** are imperative constructs, efficient compilation, and ease of implementation of dataflows, which are crucial in control and signal processing applications.

(iii) SystemC is a C++ based language for system modeling, that implements a discrete-event computation model. The language provides libraries that specify processing modules and specify input-output ports for communication. The language is both a description language and a simulation kernel, making it easier to transition from specification to executable implementation. LabVIEW offers some important advantages compared to System

Table 1: Hybrid language features compared. Full support •, and partial support ◦. Part of the table in [13]. CCSS refers to CoCentric System Studio, imp. to implementation and mgmt. to management. *Support in LabVIEW through timed loops.

Features	Esterel	SDL	SystemC	CCSS	**G**
Concurrent	•	•	•	•	•
Hierarchy	•	•	•	•	•
Preemption	•		•	•	•
Deterministic	•		◦	•	•
Synchronous commn.	•		•	•	•
Buffered commn.		•	•	•	•
FIFO commn.		•	◦	•	•
Procedural	•	◦	•	◦	◦
Finite-State-Machines	•	•	◦	•	•
Dataflow		•	•	•	•
Multi-rate dataflow			◦	•	•*
Software imp.	•	•	•	•	•
Hardware imp.	•		•	•	•
Memory mgmt.	◦	•			•

C: automated and efficient memory management, which prevent memory leaks, and a dataflow-based environment.

(iv) CoCentric System Studio [16] is a hierarchical specification language that combines PNs, FSMs, and gated models. Dataflow block primitives can be written in C++. The development environment offers a simple simulation and debugging mode, as well as support for code generation. Compared to CoCentric modeling, the main advantage of LabVIEW is that the simulation and execution platforms are the same, and automated memory management. Signal processing and control libraries are available to both.

(v) Simulink is a system level description environment [17], based on the computational algebra tool Matlab. Simulink has a real-time code generation module Real-Time Workshop, that allows described systems to be downloaded to specific real-time boards. Interactivity and online experimentation are very complicated in this environment since each time a real-time program is to be run, it needs to be recompiled. The user interfaces are also not easy to create and modify as compared to LabVIEW.

(vi) Some other languages that are used, but are much less popular, are PMC and Forth [18]. PMC is based on a mathematical abstraction called process algebra. It does not support object orientation nor dynamic allocation of memory, but has primitives for synchronization. Forth is an imperative interpreted language, which supports a variety of processors.

4. EMBEDDED MOTION CONTROL

Modern motion systems, such as robots, involve a variety of different elements that come together in a complex system. On the one hand, they are constructed from electromechanical components that combine mechanical elements such as wheels and gearboxes with electrical components such as power converters, digital circuits, and solid-state sensors. The operation of these electromechanical systems is choreographed by embedded software programs that abstract the dynamics of these interacting elements and provide a basis for higher level programs that perform reasoning and deliberation. For instance, the mechanical components are subject to physical constraints, the electrical components involve a variety of transient dynamics and the communication channels between low-level elements involve significant uncertainty. The role of the low-level embedded software programs is to hide this complexity and provide simpler symbolic or logical variables with which to deliberate at the higher level.

These notions of hierarchy and modularity represent one of the best ways to overcome system-level complexity, in a variety of different domains [19]. However, the way these notions have been brought to bear on typical motion systems has involved difficult tradeoffs.

In many robots, including some fairly modern examples such as Sony's AIBO and its many clones, the joint-level control modules handle all of the above mentioned complexity and expose a limited set of parameters to the higher level programs. So, even though the joints are equipped with servo motors and sophisticated circuity that could—in principle—support many different model-based and adaptive control strategies, what the user actually gets is a PID controller with limited control over setpoint, rise time, and such variables. This has the important implication that task-level control algorithms are now restricted to only dealing with kinematic variables and not with the true dynamics of the robot. Moreover, this approach leads to algorithmic limitations at the higher level (e.g., in the expressiveness of the planning and control strategy) that have little to do with the task at hand and much more to do with architectural decisions regarding the embedded system implementation.

In response to these limitations, there is growing interest in developing tools and techniques that can span the entire space from high-level specification (such as "go from *A*

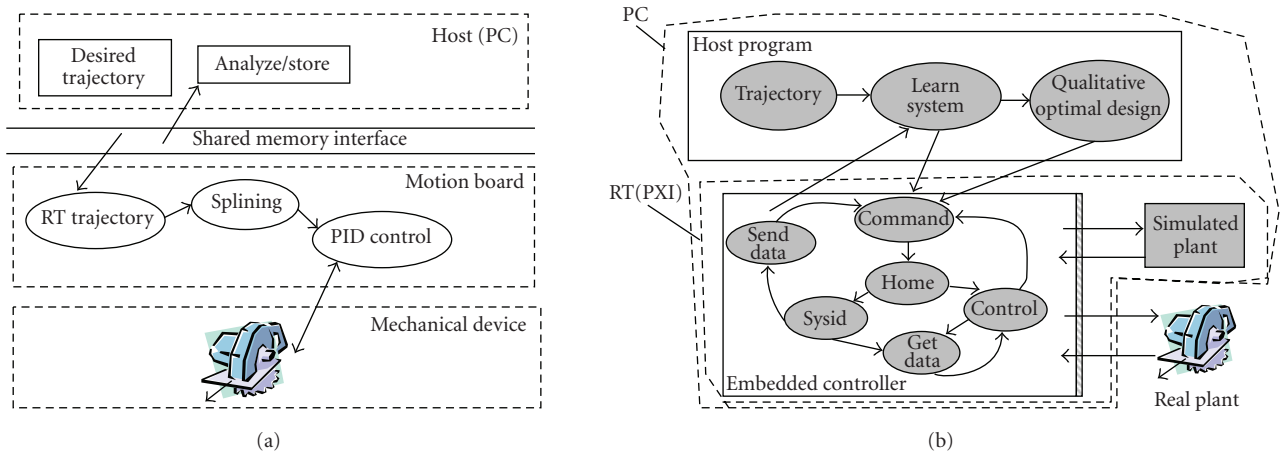


Figure 2: (a) Commercial motion control solutions and (b) novel hierarchical solution with hardware-in-the-loop components and true plant.

to B without bumping into trees and vehicles") to low-level specifications (such as limits on response time of an actuator). Researchers have begun to actively explore the deliberative and algorithmic aspects of this problems, for example, [20]. However, the execution and implementation side of this problem is not well explored.

In this section, we outline an architecture for a motion control system that addresses these issues. We will begin with a general discussion on hierarchical architecture of motion control components—of sufficient complexity to handle the sorts of difficult requirements mentioned above. Then, we identify a problem instance that captures the essence of this problem and we discuss how our proposed framework can address them. We will show that it is possible to map the hierarchical nature of the algorithmic problem-solving process onto corresponding components in an embedded system tool chain.

We have already showed that following simple design principles, LabVIEW and **G** are a competitive and sound language for embedded system development, simulation, and implementation. The availability of LabVIEW, LabVIEW-RT, and the hardware support for sensing and control with platform-independent execution, makes this environment ideal to map the hierarchical components of our problem. Components that require real-time response are mapped into **G** programs compiled by LabVIEW-RT and downloaded to a real-time board. Components that require user interface and are not real-time critical are run in a standard computer, in LabVIEW. From the developers standpoint though, program development and deployment are transparent since no complicated compilation procedures, nor a different environment for development and deployment are required.

4.1. A flexible architecture for embedded motion control

Embedded motion control as implemented in the industry today is primarily based on the Proportional Integral Derivative (PID) control algorithm. Well over ninety percent of existing controllers involve PID controllers that control each axis of the system independently [21].

A typical motion control system is presented in Figure 2(a). The host machine handles the user interface and higher-level executive routines. The trajectory generator outputs position versus time data on the fly. This data is used by a PID control loop to drive the plant, which could be composed of several motors (typically, one for each axis). In existing systems, the control loop is programmed on the board and can only be changed by rewriting embedded code. The key limitation of this approach is the lack of flexibility in terms of algorithm and parameter ranges.

Commercially available motion control systems push the limits of currently available real-time boards and software. Yet, they are sometimes inadequate for applications in areas such as robotics, aerospace, and other high-performance systems. The limitation of existing systems is that their architecture makes it hard to offer control strategies that require dynamic adaptation [21]. For example, deployment of sophisticated control strategies for high accuracy, such as multiple axis multiple-input multiple-output (MIMO) control, requires a precise identification of the system under control. In many cases, optimal implementation lacks flexibility and tends to require fairly sophisticated hardware design techniques.

We propose a different approach to the problem, enabled by the availability of LabVIEW-RT and **G**. Consider the hierarchical motion control system described in Figure 3. The key elements in this architecture are as follows:

(i) System adaptation: the structure contains an online system identification component that allows the whole structure to adapt to the motors currently being used. It also includes control design calculations for the current system. This requires a two-way communication between the hierarchical levels.

(ii) Task level Planning: this block represents all the deliberation that goes into defining a concrete trajectory from the domain-specific requirements of the task. The

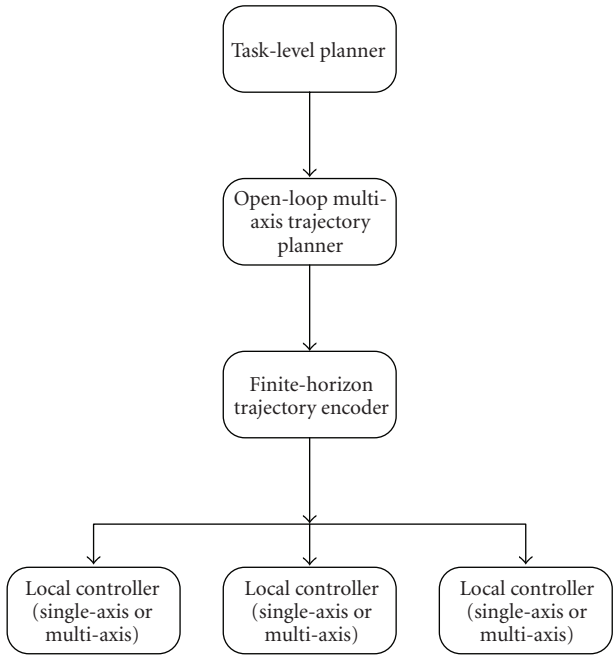


Figure 3: A hierarchical motion control architecture.

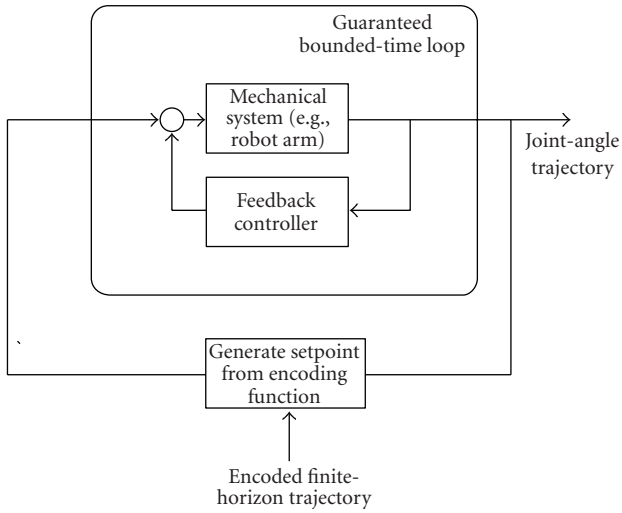


Figure 4: Architecture of the local control module—which could be a single axis or a set of local axes.

output of this block might take the form of specific way points that the motion system must achieve, for example, a set of points, $\hat{x}_i$, or a time-indexed set of points $\{(\hat{x}_i, t_i)\}$. The process of implementing this deliberation often requires discrete models of computation such as finite state machines and other discrete-event structures (e.g., petri nets).

(iii) Open-loop trajectory planing: a motion system will need a concrete specification of where it should be at all times. So, the nonuniformly spaced (time-indexed) points must be interpolated in an appropriate way in order to obtain a uniformly spaced sequence, $\{(x_i, t_i)\}$. One way to achieve this interpolation is to use multidimensional splines (see below).

(iv) Finite-horizon trajectory encoder: it is typically easier to deal with disturbances when the trajectory is specified in an intrinsic spatial way rather than extrinsically in terms of a time index. With this in mind, one might wish to describe the desired trajectory in terms of some functional form, such as a polynomial. Also, in order to permit higher-level replanning if significant errors are detected, this is done only over some finite horizon.

(v) Low-level control: with the above setup, it becomes possible to pose the low-level problem in such a way that a feedback controller can take in a finite-horizon specification for a subset of the available axes of motion and ensure stable execution in the face of sensor noise and other disturbances. The structure of this block is depicted in Figure 4.

In order to be flexible, such an architecture must support the following features:

(i) ability to incorporate constraints at any level in the hierarchy, without significantly restricting the degrees of freedom of other levels;

(ii) clear characterization of the region of operation for each level;

(iii) the ability to go from the specification of each level to executable code running on the corresponding embedded platform.

These requirements imply a number of things for both planning and control algorithms and the embedded system implementation. Considering that the low-level control modules do not have direct access to the global state, hence the state of many other parts of the system, it is desirable to have online algorithms for model identification and control design. At the higher level, the task requirements may continually change. So, it would be desirable to have mechanisms for tuning the dynamic properties of lower-level modules based on high-level requirements. Correspondingly, lower-level components need the ability to modulate the operation of higher-level planning operations. Although this is conceptually simple, few embedded system implementation frameworks support this upward flow in a seamless manner.

In the remainder of this paper, we will illustrate such a motion control system—explaining the design and implementation of these various components in the context of a two-axis system. Having already described the properties of our basic programming language models, we will show here that it is indeed possible to map these algorithmic components onto suitable data-flow programs. In this sense, we will outline a novel framework for *interactive* motion control. Among other things, this framework can also support the hardware-in-the-loop paradigm.

4.2. Online implementation of system identification and control components

4.2.1. System identification

Model-based control design begins with the definition and identification of a suitable model of the dynamics of the system. Often, there is insufficient information available to derive these models form first principles. Instead, the preferred approach is to observe the behavior of the system, in terms of input-output data, and infer the underlying model through the algorithmic process of system identification. For DC motors, the system model in Laplace s-domain

$$P(s) = \frac{K_m}{s^2 + T_m s} \tag{1}$$

captures the physical behavior appropriately, where the parameters can be physically interpreted as a signal gain, K_m, and time constant, T_m. The parameters can be identified following a procedure that minimizes the error between the observed output y and the estimated output $\hat{y}$ for a given input x [22, 23]:

$$J^* = \min_{T_m, K_m} \sum_{k=1}^{N} \left[\hat{y}(k) - y(k)\right]^2. \tag{2}$$

The model derived for a single axis can be naturally generalized to multiple axes. For most usual systems, axis transfer functions can be identified independently.

The optimization in (2) can only be solved if the system being identified is stable. In our approach, we use a simple stabilizing controller for each axis. For the single axis, a simple proportional gain controller can always stabilize it, albeit inefficiently. The plant can be identified using this controller. Mathematical details can be derived following techniques in [12].

4.2.2. Control design

A controller architecture that satisfies our requirements of flexibility and interactivity is the linear quadratic regulator (LQR). This controller can be used for a single-axis or for a local channel of multiple axes, and is compatible with the scheme depicted in Figure 4. In this section, we will briefly describe this controller and we will specifically point out how this structure supports interlevel communication: (a) in allowing a higher-level module to adjust the dynamic properties and (b) in indicating that the system has exited its region of applicability and higher-level replanning is necessary. We build our multiple axis system model from the identified parameters using a linear state space system model

$$\dot{x}(t) = Ax(t) + Bu(t), \tag{3}$$

where $x(t) \in \Re^n$ and $u(t) \in \Re^m$ are the state and control actions , respectively. The control task, that of executing a given finite-horizon trajectory segment, can be defined by a cost function that specifies costs on deviation from this desired trajectory,

$$J = x'(T)Q_T x(T) + \int_0^T \left(x'(t)Qx(t) + u'(t)Ru(t)\right)dt, \tag{4}$$

where Q_T, Q and R are symmetric positive definite matrices representing costs. A higher level module is free to set these matrices separately for each trajectory segment—and in this sense, the problem is flexible. Using optimal control [12], the above problem can be solved by using an input $u(t) = -R^{-1}B'K(t)x(t)$, where $x(t)$ is the observed state. The gain function $K(t)$ is calculated solving a nontrivial optimization problem, and depends on the choice of the matrices Q, Q_T, and R.

In particular, the gain matrices can be parameterized in simple ways such that intuitive concepts such as *stiff* of *smooth* response may be mapped onto them. For instance, consider the matrix form $Q = \text{diag}(a_1, a_1, a_2, a_2)$ where all off-diagonal elements are zero. The parameters a_1, a_2 can now control the relative importance of trajectory following and minimum effort. This is appealing to a typical user of industrial motion systems.

The LQR controller represents an "automatic" design procedure that is only valid within a specific region of applicability, which can be computed using sophisticated approximation methods [24]. So, for instance, if we know the maximum energy that can be applied by an actuator, then we can restrict the region of operation and the lowest level module is now in a position to influence a higher-level planning module by indicating an exit from the safe region. This is a desirable aspect of the flexible architecture.

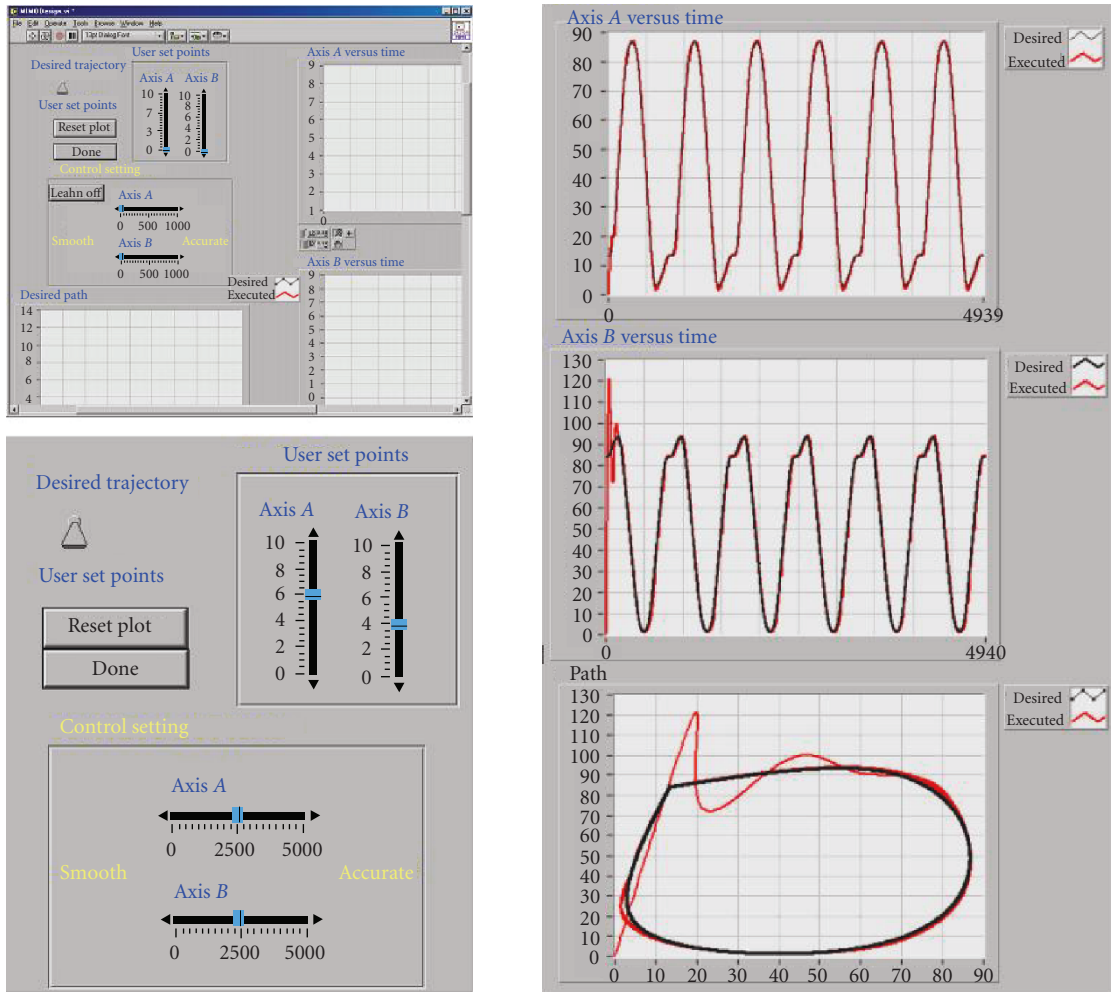
4.3. Integration of design components

In this section, we map the proposed hierarchical system model and online algorithms to an actual embedded system for generic motion control. When compared with existing motion control architectures, the LabVIEW RT-based system provides an easy way to use, flexible control environment.

Our system encapsulates a mechanism for the user to start with unknown plant parameters, identify the parameters experimentally, design a control algorithm, and run it on real-time hardware in one seamless system. Furthermore, control design can be executed in real time with effects of varying cost choices observed immediately in system response. This level of design integration is new in embedded motion control.

The software-hardware mapping of the new architecture is shown in Figure 2(b). Compare this to existing commercial architectures shown in Figure 2(a). The host program runs in a desktop PC. The real-time embedded controller runs in a commercial PXI computer system, with LabVIEW RT and RTOS installed. The embedded controller and host program are completely implemented in **G**. The host computer and the PXI system communicate via a TCP/IP-based protocol. This enables the RT subsystem to be used anywhere in a distributed system and still be controlled by the host.

We start by describing the host software. A straightforward user interface has been implemented for the system (Figures 5(a) and 5(b)). It allows the user to start with a qualitative plant model (in fact can be more general than the DC motor), identity unknown plant parameters, design and implement a control strategy. Furthermore, a high-level desired trajectory can be specified for a designed system.



(a) (b)

FIGURE 5: Interface of the real-time adaptive motion control system showing (a) host system control panel and details and (b) a running system after learning the controller.

The host PC handles the interpolation and trajectory generation. This structure allows uninterrupted operation of the controller while still allowing complete flexibility in operation. Notice that the expensive trajectory generation and refinement operations could disrupt the real-time operation of a system. The information that is sent to the RT board is the array of set points.

There are two methods of specifying the desired trajectory. It is possible to specify a few points in the path and allow the program to complete the path, based on cubic spline interpolation. The other method is to use a slider on the screen to directly specify the set point. This could be used to interactively observe system response to arbitrarily chosen set point changes. Trajectory generation also accepts specification of upper limits on velocity and acceleration. Once the trajectory has been specified and converted to an array of set points, the next step is to identify plant parameters. The user can allow the program to do this for him or specify plant parameters to be used in the control.

In order to identify the plant parameters, the trajectory is sent to the RT system and the embedded controller is commanded to begin system identification. The RT system runs an assumed proportional gain algorithm to make the system follow the desired trajectory, using the previously defined points as set points. The actual path followed by the system and the desired trajectory is returned by the RT system, and used by the host system identification program to determine plant model parameters.

Once the plant parameters have been identified, the user can choose to control the motors at any time for desired regular operation. The choice is made at the host program and communicated to the embedded system.

We can now describe the embedded controller. The controller is implemented in **G** using a combination of finite state machines (FSMs) and dataflow programs [25]. The FSM implements the state machine as in Figure 2(b). For the end user, such an implementation allows for adaptability. Different control strategies could be programmed and easily

added as a new state in the controller. The communication and input/output infrastructure would remain the same.

Currently, the FSM in the controller can be directed to run either a simple gain algorithm for system identification or a MIMO feedback algorithm, with gains chosen by the interactive control design program in the host PC. The choice is dictated by a command from the host program. The FSM responds to requests that arrive via a TCP/IP queue from the host program. The FSM architecture allows multiple control algorithms to be stored in the embedded controllers allowing any one to be immediately invoked when commanded.

In our hierarchical approach, the desktop PC handles all user interface-oriented functions and the embedded controller handles the stringent real-time control loop. The two processes are capable of communicating with each other. However, the absence of the communication link does not affect the operation of the control loop. This brings an element of fault tolerance to the system.

A major feature of the proposed interface and system is that the user can dynamically modify control parameters and experiment with the plant response. This allows multiple control strategies to be compared in a "what-if" simulation or even in real-time execution. The host program also implements a simulation of the estimated plant, so that the system response can be simulated before being sent to motor control. When the user changes to control parameter seem satisfactory in simulation, this data can be immediately relayed to the embedded controller.

We have implemented an intuitive slider-based control design. The slider ranges from smooth to accurate. In smooth mode, the control response is slower, but there are no overshoots when following trajectories. In the accurate mode, response is faster, and depending on the choice, overshoot can happen. The interesting feature here is that the response of the whole system is fast enough that moving the slider, one can immediately see the effects on a real system, if this is desired.

Another important characteristic of the system is that the structure allows the real plant to be replaced by a simulated plant in software without requiring significant modifications. This technique (called "Hardware in the loop" abbreviated HIL) [26] is a popular system verification and simulation tool. Traditional systems do not have such a well-defined and easy-to-use interface to support HIL experiments, especially for user-defined algorithms.

Currently a two-axis controller running at frequencies up to 10 KHz is supported by the system. It is expected that such high frequencies are attainable even with more sophisticated control structures and algorithms.

5. EMBEDDED CONTROL DESIGN USING LabVIEW RT

Some important requirements for successful development environment for embedded programs in control are as follows:

(i) facilitate development of reliable programs;

(ii) simplify integration of different levels of a hierarchy, making it easy to map into different platforms (e.g., real time, FPGA);

(iii) availability of standard control design and signal processing routines;

(iv) source code needs to be easy to maintain;

(v) code should be easy to read and interpret;

(vi) parallelism and concurrency are easy to represent.

The proposed system description language and design environment address all these issues as shown in earlier sections, as long as simple design principles are followed.

Previous works in COSSAP [27], GRAPE [28], Simulink [17], and Ptolemy [29] have shown the importance of using higher-level representation constructs to build real-time functionality. The applications there have focused on multirate signal processing, with online adaptation and learning for a variety of tasks such as channel equalization and echo cancelation [30]. The transparent transition from a simulation environment to a real deployment has been studied for such problems.

Control programs on the other hand are typically hierarchical as explained in Section 4. The motion control application developed in this paper demonstrates two novel possibilities: the mapping of hierarchical control structures into an embedded execution platform, and the handling of a variety of different problem time scales in a transparent manner. The later capability in fact shows an interesting new direction for control systems design, where even more complicated design optimizations are done in real time, based on observation of current system performance, in a typical hierarchical feedback loop. The design of the **G** language and the integration level of the LabVIEW-RT environment were helpful to achieve these goals.

5.1. Design process

In Figure 6, we compare the traditional design process with the new design process that becomes possible as a result of our system architecture. We show this comparison in terms of the example application developed in this paper.

In the traditional design process a model of the system is used to devise a possible control strategy. This is tested with a computer-based simulation and then ported to a prototype hardware test rig. This process is iterative and each iteration leads closer to the final product. The key issue here is that the design, simulation, testing, and integration are done using an array of loosely coupled tools. There are no tightly integrated system design environments that allow the designer to simulate and test the design iteratively without having to move between multiple programs, maybe even multiple platforms. Recent advances in hardware software codesign address the problem of optimum design of a subsystem. They do not address the larger issue of integrating dissimilar subsystems in a time and cost optimal manner.

A design generated in the LabVIEW RT environment is directly used for hardware-in-the-loop simulations where

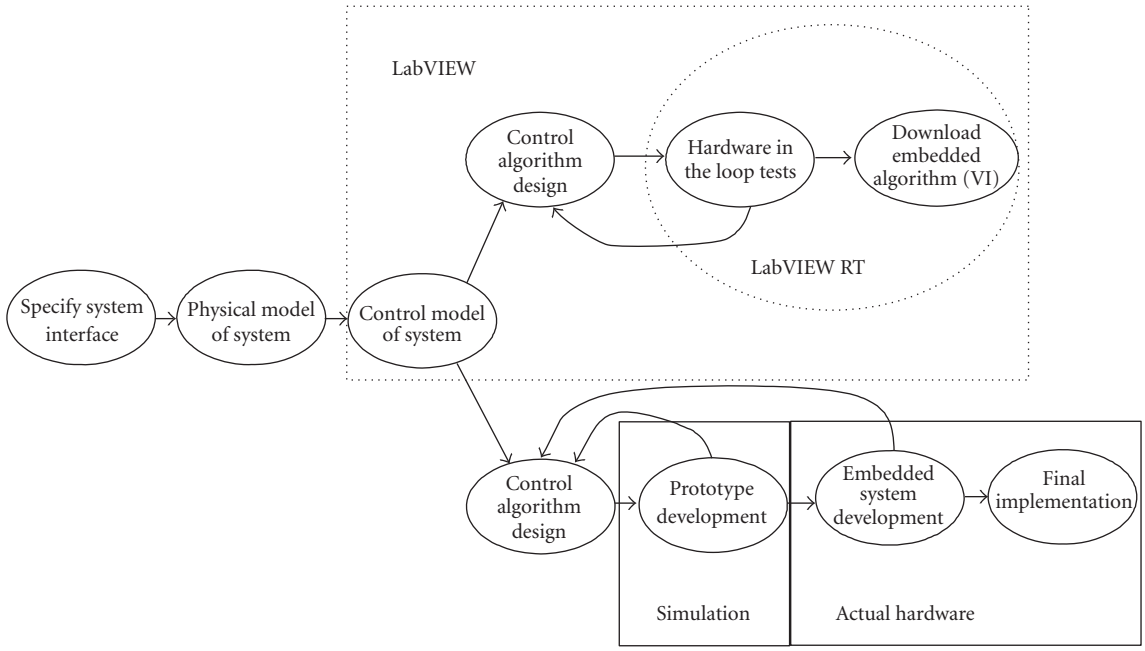


Figure 6: Hierarchical real-time control design flow.

the controller is tested in the environment where it might eventually be used. The design, simulation, and test environments are very tightly coupled and in many cases are transparent to the user. This reduces the need for compromises based on compatibility considerations and provides the designer with many more degrees of freedom, leading to significant reductions in the development time and in the cost of the test cycle.

From the design of the motion control solution, we also notice that an important principle is that lower levels in the control hierarchy are implemented in real-time environments, and higher levels are implemented in systems with intensive and intuitive user interfaces. Usability is an important and overlooked issue in most control systems, due to the difficulty of achieving it without hierarchical design and implementation.

Furthermore, easy integration of the different hierarchical components allows for data and control flow in both directions in the hierarchy: from top to bottom, carrying design specifications and high-level commands, and from bottom to top, providing information that can be used for interactive tuning of the system and system identification.

Another important characteristic that is useful in embedded control design is that the control design algorithm itself should be supported by the embedded environment. This enables high performance adaptive control, but also implies that the embedded description language and environment should have a certain level of sophistication for such implementations to be easily developed and deployed.

6. CONCLUSIONS

In this paper we explored the application of LabVIEW RT for embedded software development. We showed that **G**, the underlying model of computation, satisfies the requirements for specifying embedded systems and with certain simple design principles it becomes a sound language for embedded development.

We developed an embedded motion control system as an application of LabVIEW RT for embedded design. The main advantage that LabVIEW RT presents is a reduction in development time because of the combined simulation and execution environments. Another important advantage is that **G** allows us to easily combine different models of computation.

The implemented motion control system includes automatic plant modeling (system identification), MIMO control capabilities, and an innovative qualitative optimal control design program. Moreover, our architecture implements HIL techniques for replacing real plants with software simulated plants. The system is also very flexible, allowing the end user to access and replace control routines.

This work can be extended to include improvements in the qualitative control design methodology that will allow for more generic control algorithm specifications. Moreover, the current system identification can be extended to identify systems with coupled axes.

We showed how a hierarchical system design can be directly mapped to a real-system implementation, with novel results in terms of capabilities. The processing performance

of the system is on par with many current commercial systems, but with much increased flexibility and control performance.

The hierarchical approach addressed in this paper will become even more important with current and future embedded applications. The hierarchical embedded control systems design should be supported for development and deployment from a host system running a standard operating system (OS), to a real time OS, to an FPGA.

We are currently exploring the development an application that has three layers: one in the host computer system, one in a real-time system, and a third layer in an FPGA system. We plan to use LabVIEW FPGA to deploy the FPGA system.

ACKNOWLEDGMENTS

The authors would like to acknowledge the significant improvements to the paper thanks to the suggestions of the reviewers.

REFERENCES

[1] ITU-T recommendation Z.100, "Specification and description language," International Telecommunication Union, 1999.

[2] H. Andrade and S. Kovner, "Software synthesis from dataflow models for G and LabVIEW™," in *Proceedings of the 32nd Asilomar Conference on Signals, Systems, and Computers*, vol. 2, pp. 1705–1709, Pacific Grove, Calif, USA, November 1998.

[3] C. Belta, A. Bicchi, M. Egerstedt, E. Frazzoli, E. Klavins, and G. J. Pappas, "Symbolic planning and control of robot motion [Grand Challenges of Robotics]," *IEEE Robotics & Automation Magazine*, vol. 14, no. 1, pp. 61–70, 2007.

[4] G. Berry and G. Gonthier, "The Esterel synchronous programming language: design, semantics, implementation," *Science of Computer Programming*, vol. 19, no. 2, pp. 87–152, 1992.

[5] S. S. Bhattacharyya, P. K. Murthy, and E. A. Lee, *Software Synthesis from Dataflow Graphs*, Kluwer Academic Publishers, Norwell, Mass, USA, 1996.

[6] S. P. Bhattacharyya, H. Chapellat, and L. H. Keel, *Robust Control: The Parametric Approach*, Prentice-Hall, Upper Saddle River, NJ, USA, 1995.

[7] J. Buck and R. Vaidyanathan, "Heterogeneous modeling and simulation of embedded systems in El Greco," in *Proceedings of the 8th International Workshop on Hardware/Software Codesign (CODES '00)*, pp. 142–146, San Diego, Calif, USA, May 2000.

[8] J. T. Buck, "Static scheduling and code generation from dynamic dataflow graphs with integer valued control signals," in *Proceedings of the 28th Asilomar Conference on Signals, Systems, and Computers*, vol. 1, pp. 508–513, Pacific Grove, Calif, USA, October-November 1994.

[9] T. H. Cormen, C. E. Leiserson, and R. L. Rivest, *Introduction to Algorithms*, McGraw-Hill, New York, NY, USA, 1990.

[10] A. Datta, M.-T. Ho, and S. P. Bhattacharyya, *Structure and Synthesis of PID Controllers*, Springer, London, UK, 2000.

[11] K. Dutton, S. Thompson, and B. Barraclough, *The Art of Control Engineering*, Addison-Wesley Longman, Boston, Mass, USA, 1997.

[12] S. A. Edwards, "Design languages for embedded systems," Tech. Rep. CUCS-009-03, Columbia University, New York, NY, USA, 2003.

[13] A. Girault, B. Lee, and E. A. Lee, "Hierarchical finite state machines with multiple concurrency models," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 18, no. 6, pp. 742–760, 1999.

[14] S.-H. Han, M.-H. 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, no. 2, pp. 194–204, 1999.

[15] National Instruments, *LabVIEW 7 Software Reference and User Manual*, 2002.

[16] D. E. Knuth, *The Art of Computer Programming: Fundamental Algorithms*, Addison-Wesley, Reading, Mass, USA, 1997.

[17] H. N. Koivo and J. T. Tanttu, "Tuning of PID controllers: survey of SISO and MIMO techniques," in *Proceedings of the IFAC International Symposium on Intelligent Tuning and Adaptive Control (ITAC '91)*, pp. 75–80, Singapore, January 1991.

[18] J. Kunkel, "Cossap: a stream driven simulator," in *Proceedings of the IEEE International Workshop on Microelectronics in Communications*, Interlaken, Switzerland, March 1991.

[19] R. Lauwereins, M. Engels, M. Ade, and J. A. Peperstraete, "Grape-II: a system-level prototyping environment for DSP applications," *Computer*, vol. 28, no. 2, pp. 35–43, 1995.

[20] B. Lee and E. A. Lee, "Hierarchical concurrent finite state machines in ptolemy," in *Proceedings of the 1st International Conference on Application of Concurrency to System Design (ACSD '98)*, pp. 34–40, Fukushima, Japan, March 1998.

[21] E. A. Lee and D. G. Messerschmitt, "Static scheduling of synchronous dataflow programs for digital signal processing," *IEEE Transactions on Computers*, vol. 36, no. 2, pp. 24–35, 1987.

[22] E. A. Lee and T. M. Parks, "Dataflow process networks," *Proceedings of the IEEE*, vol. 83, no. 5, pp. 773–801, 1995.

[23] W. S. Levine, *The Control Handbook*, CRC Press, Boca Raton, Fla, USA, 1996.

[24] Mathworks, *Simulink User Manual and Online Help*, 2001.

[25] S. S. Maurer, "A survey of embedded system programming languages," *IEEE Potentials*, vol. 21, no. 2, pp. 30–34, 2002.

[26] P. K. Murthy, "Scheduling techniques for synchronous and multidimensional synchronous dataflow," Tech. Rep. UCB/ERL M96/79, University of California at Berkeley, Berkeley, Calif, USA, 1996.

[27] S. Note, P. van Lierop, and J. van Ginderdeuren, "Rapid prototyping of DSP systems: requirements and solutions," in *Proceedings of the 6th IEEE International Workshop on Rapid System Prototyping (RSP '95)*, pp. 88–96, Chapel Hill, NC, USA, June 1995.

[28] T. M. Parks, "Bounded scheduling of process networks," Tech. Rep. UCB/ERL-95-105, University of California at Berkeley, Berkeley, Calif, USA, 1995.

[29] J. L. Pino, S. Ha, E. A. Lee, and J. T. Buck, "Software synthesis for DSP using ptolemy," *The Journal of VLSI Signal Processing*, vol. 9, no. 1-2, pp. 7–21, 1995.

[30] H. Simon, *The Sciences of the Artificial*, MIT Press, Cambridge, Mass, USA, 3rd edition, 1996.