Open Source, Simple, Concurrent Simulator for Education and Research

Miguel Bazdresch
ECTET Department
Rochester Institute of Technology
Rochester, NY, USA
Email: miguelb@ieee.org

***Abstract*—In engineering education and training, it is desirable that students develop their own simulators, or inspect the source code, modify it, and design their own functionality based on an existing simulator. Existing commercial and open-source simulator software are not always appropriate for this purpose, given their complexity or their closed-source nature. We propose a simulator that is simple enough to be used for this purpose, while offering powerful features (such as concurrent computation and graphical user interfaces) that make it adequate also for more advanced research work.**

***Keywords*–Modeling, Computer Simulation, Parallel Processing, Educational Technology**

I. Introduction

In engineering education or training, it is often desirable that students build their own simulators, inspect and/or modify the source code of an existing one, or design their own code and insert it into a simulator. It is difficult to accomplish these objectives in the context of an engineering course, with a reasonable time investment on the part of the students. It would be useful to have an educational simulator with the following features:

- Open-source, so that its code may be freely inspected and modified.
- Coded in a simple, dynamic language.
- With high performance.

In this paper, we present a simulator that meets this description. The simulator is similar to Simulink [1] or LabView [2]. It allows simulation of models that may be divided into independent, interconnected blocks. Data and/or events flow among the blocks, of which some are sources, some sinks, and some process their input into an output. It may be useful for discrete simulation, for model-based design, for implementing stencil codes and for simulating other types of distributed models.

The paper is divided as follows. In Section 2, we present an introduction to the Julia language; the simplicity and performance of the proposed simulator are, in large part, due to this language. In the third section, we describe our proposal in some detail. Then, we compare it with other existing simulators offering similar capabilities. We finish the paper by presenting our conclusions and our plans for the future.

II. The Julia Language

Julia [3], [4] is a recent open-source numerical computing language. It offers the simplicity and expressiveness of more well-known dynamic languages such as Python [5] or Matlab [6], with execution times that are frequently no more than two times slower than C or C++ code. In comparison, algorithms coded in Python or Matlab often are thousands of times slower than their equivalents written in C.

Julia achieves this unprecedented (for a dynamic language) execution speed thanks to a “just in time” compilation strategy, based on the Low-Level Virtual Machine (LLVM) [7].

The combination of speed and simplicity is one of Julia’s main attractions; however, it has other features that make it interesting as a simulation programming language: its typing system and its built-in parallel and distributed computing features. Julia’s development is led by a team at the Massachusetts Institute of Technology. Although still in its early stages, it has attracted considerable interest. It is available for the three major operating systems, Windows, Linux and OS X.

A. Julia’s typing system

Julia’s typing system is powerful and flexible. It offers a nominative type hierarchy with bits (numbers), composite (structures) and dictionary types, among others. Types may be used as arguments. The compiler creates efficient code by using run-time, dynamic type inference on variables. Julia allows the option of declaring (annotating) a variable's type, which helps the compiler produce faster code.

In the context of simulation, besides its execution speed benefits, a powerful type system helps to insure that interconnected blocks receive and produce data of the expected types, avoiding run-time failures and coding errors.

B. Julia's parallel and distributed computing environment

Multi-core, distributed, clustered, cloud and otherwise parallel and concurrent computing platforms promise significant gains in simulation speed [8]. While parallel programming is not a trivial task, it can be made more tractable by appropriate support from the programming language. Julia provides a multiprocessing environment based on message passing. Each processor runs its own Julia code on its own memory, and all data sharing is done explicitly by passing messages between CPUs.

Multiprocessing lends itself to simulating a model which can be divided into independent blocks, each of which operates on data produced by other blocks and hands its data to other blocks. Each block runs independently on each processor, and exchanges data with other blocks using message passing. The main drawback of this technique is the overhead involved in passing messages.

Julia also supports automatic compilation for multiprocessing and cluster computing. For instance, arrays may be distributed, in the sense that it is divided into subarrays, each existing on a different CPU's memory. Operations on the array are performed by all CPUs in parallel, behind the scenes and with very little explicit intervention by the user.

III. The proposed simulator

In this section, we describe a simulator that takes advantage of Julia's strengths, and that is sufficiently simple to be adopted in an educational context, while being powerful enough to be useful in research.

Assume that we wish to simulate a model that has been divided into a number of discrete blocks. Each block has several input and/or output *pins*, each of which is connected to one or more pins of other blocks.

In a concurrent simulator, a scheduler assigns blocks to available computing resources according to certain rules. We now describe our proposal in more detail.

A. Blocks

A block is a structure with the following fields:

- A configuration, implemented as a hash table. Each block defines its own configuration and how it is interpreted.
- A *work* function, which processes the inputs to produce the outputs.
- A *stop* function, which is called when the scheduler stops the simulation.
- A *state*, where the block may store its state.
- A set of input and/or output pins, each of a given type.
- A flag that tells the scheduler that the simulation should be stopped.
- A flag that tells the scheduler that the block's work function is currently being executed.

Some configuration items may specify how the block is to be executed and scheduled. For example, if a block opens a file, then the file descriptor will be valid only for the CPU where the file was opened. In such a case, it is required that the block is executed always in the same CPU. As another example, multiprocessing (in contrast to multithreading) makes it easier to manually partition work among the available CPUs. This may be achieved by indicating to the scheduler that a block should only be executed on a set (or subset) of the available processors.

After blocks are defined, they are instantiated. A given block instance has its own configuration and state, which may be different from those of other instances. Once blocks are instantiated, their pins are connected using a simple function. In this way, a dataflow graph is built, with data pipes connecting the blocks.

B. Scheduler

The scheduler executes in a single processor. It traverses the dataflow graph, trying to find a block that meets a set of criteria for execution. Once a block is found, it calls the block's work function on a free CPU. Then, it continues the search.

A block that meets all the following criteria is eligible for execution:

1) The block must not be running on another CPU.
2) If the block specifies execution on a particular processor, then said processor must be free.
3) All of the block's input pins must be connected to pipes that are not empty; likewise, all its output pins must be connected to pipes that are not full.

This scheduler is very simple, but it allocates work in an efficient manner and, if there are more blocks than CPUs, it is able to keep all CPUs busy.

If it is desired, it is very easy to modify the scheduler or create a new one, due to the simplicity of the language and its dynamic nature.

C. Interfaces

The simulator supports two kinds of interfaces. The first is support for graphical user interfaces. These are blocks that represent data graphically on the screen, and also allow the user to interact with the simulator via traditional user interface elements such as buttons or menus. One example would be an oscilloscope block that displays a signal. Another example would be a waterfall display.

The second kind of interface blocks are those that connect to external hardware elements and allow interaction with physical signals or data. This allows the model to be tested with the same kind of data an eventual system implementation would interact with. For instance, we provide source and sink blocks that connect to a computer's sound card, which in this case acts as a general, low-bandwidth signal acquisition and generation system.

IV. Example

We present an example of a simple block. The block takes as input a vector of floating point numbers. Its output is the square of the input's elements. This block has one input pin and one output pin. We need to create a function to instantiate the block:

```
function square_float_inst()
  b=Block({},                 # empty dictionary
    x::Vector{Float}->x.*x    # work function
    ()->nothing               # stop function
    0,                        # initial state
    [Pin(Vector{Float})]      # input pin
    [Pin(Vector{Float})])     # output pin
  return b
end
```

In this code, `Block()` is a constructor that returns a block with the given arguments, and `Pin()` returns a pin of the given type. Note we can use types as constructor arguments in a very simple way. We use an empty dictionary as configuration, since this block has no configurable options.

In this case, the stop function is an empty function. Using anonymous functions, this is specified as `()->nothing`. The work function is defined also as an anonymous function, but note that in this case we used type annotations to ensure correctness and to help the compiler. For more complex cases, the work function can be defined separately.

The user would create a block instance `squarer` by calling `squarer = square_float_inst()`, and would connect its input pin to an already instantiated source block `source` with `connect(source,1,squarer,1)`. With this code, output pin 1 of the source block is connected to output pin 1 of the squarer block. The function `connect()` verifies that pin types match, and builds the simulation dataflow graph. With the function `run()`, the scheduler launches the simulation.

V. Comparison with similar simulators

Our proposal is similar to tools such as LabView, Simulink and GNU Radio. Of these, only GNU Radio [9] allows code inspection and modification. For this reason, we focus on it for comparison.

On one hand, GNU Radio is a much more mature tool, with many well-tested, high-performance blocks available. It is focused on modeling signal processing, software-defined radio and telecommunications systems. It offers real-time performance in some applications. It has interfaces to many kinds of hardware. It offers multithreading parallelism.

On the other hand, GNU Radio is a complex system. The blocks and scheduler are written in C++ to achieve the required performance, while the dataflow graph is defined using the Python programming language. This mixture of languages, together with other decisions in the simulator's design, introduce some level of complexity. This is exemplified in [10], a tutorial to write a block that squares its input. The block's code consists of more than 200 lines of C++, and its implementation requires the knowledge of concepts such as virtual functions and inheritance. While this complexity also provides flexibility and robustness, it may also preclude the study of GNU Radio in some educational environments. In comparison, the example presented in the previous section achieves essentially the same functionality in less than ten lines of code, and involves much less complex programming concepts.

Our proposed simulator uses a single, simple language (Julia), which offers performance competitive with C++ and C. The language's expressiveness means that inspecting the scheduler's and any block's source code is feasible within the scope of beginner or intermediate engineering courses. This is achieved without sacrificing features; on the contrary, the parallel and distributed computing facilities offered are often superior to the alternatives.

In Table I, we summarize the comparison between our proposal and GNU Radio.

TABLE I
OUR PROPOSAL COMPARED TO GNU RADIO

	Proposal	GNU Radio
Mature	No	Yes
Large library of blocks	No	Yes
Single language implementation	Yes	No
Concurrency	Multiprocessing	Multithreading
Distributed computing support	Yes	No
Code complexity	Low	High
Interface to external hardware	Yes	Yes
Graphical user interfaces	Yes	Yes

VI. CONCLUSION AND FUTURE WORK

We have presented a simulator simple enough to be used, inspected and modified by undergraduate engineering students. Even though it is simple, it offers performance that in many cases is much higher than, and at least comparable to, Matlab's, Simulink's, or Python's. This is possible due to its use of the Julia programming language. It also offers advanced multiprocessing and distributed computing support. It supports graphical user interface blocks and interfaces to external hardware.

There is much work to be done in the future:

1) The available block library needs to grow and mature.
2) Blocks that interface to a variety of hardware and computer ports have yet to be developed.
3) We wish to have an interface to blocks written in different languages, from LabView's G to Verilog.
4) The scheduler's performance has to be optimized after being measured under a variety of workloads and with different computing resources.
5) Other scheduling algorithms should be proposed and evaluated.
6) Discrete-event simulation has yet to be implemented. This may be achieved by an appropriate scheduler and by creating block pins of a time-related type.
7) The simulator has to be tried in a classroom environment, to verify its educational value.
8) In order to enable more complex scenarios, a formalism such as DEVS [11] will have to be implemented. A comparison with the large number of DEVS-based frameworks (in multiple languages) will be then possible.

We believe the Julia language, although still in its early stages, has the potential to become the premier numerical computing language. Its speed and flexibility will undoubtedly allow it to enter into computing areas now dominated by other languages, including simulation.

The simulator's source code is available for browsing and download at [12].

REFERENCES

[1] "Simulink R2012b documentation center," MathWorks, Retrieved: Nov. 2012. [Online]. Available: http://www.mathworks.com/help/simulink/index.html

[2] "Labview technical resources," National Instruments, Retrieved: Nov. 2012. [Online]. Available: http://www.ni.com/labview/technical-resources/

[3] J. Bezanzon, S. Karpinski, V. Shah, and A. Edelman, "Julia: A fast dynamic language for technical computing," in *Lang.NEXT*, Retrieved: Nov. 2012. [Online]. Available: http://julialang.org/images/lang.next.pdf

[4] J. Bezanzon, S. Karpinski, and V. Shah. (Retrieved: Nov. 2012) Julia Project, software and instructions. [Online]. Available: http://julialang.org/

[5] "Python documentation," Python Software Foundation, Retrieved: Nov. 2012. [Online]. Available: http://www.python.org/doc

[6] "Matlab R2012b documentation center," MathWorks, Retrieved: Nov. 2012. [Online]. Available: http://www.mathworks.com/help/matlab/index.html

[7] C. Lattner and V. Adve, "LLVM: a compilation framework for lifelong program analysis transformation," in *Code Generation and Optimization (CGO 2004), International Symposium on*, Mar. 2004, pp. 75–86.

[8] C. Vecchiola, S. Pandey, and R. Buyya, "High-performance cloud computing: A view of scientific applications," in *Pervasive Systems, Algorithms, and Networks (ISPAN 2009), 10th International Symposium on*, Dec. 2009, pp. 4–16.

[9] "GNU Radio Project, software and instructions," Eric Blossom and others, Retrieved: Sept. 2012. [Online]. Available: http://www.gnu.org/software/gnuradio/

[10] E. Blossom. (Retrieved: Sept. 2012) How to write a signal processing block. [Online]. Available: http://www.gnu.org/software/gnuradio/doc/howto-write-a-block.html

[11] B. P. Zeigler, "Hierarchical, modular discrete-event modelling in an object-oriented environment," *Simulation*, no. 5, pp. 219–230, Nov. 1987.

[12] M. Bazdresch. (Retrieved: Nov. 2012) Proposed simulator project, software and instructions. [Online]. Available: https://bitbucket.org/mbaz/chango