Scriptable Systems: Towards Interactive Testing during Programming of Distributed Embedded Systems

Pai H. Chou
Center for Embedded Computer Systems
University of California, Irvine, CA 92697-2625 USA
phchou@uci.edu

Scripting languages have gained popularity in the last decade as a higher level way to program computer systems. Many CAD engineers routinely converse in scripts in C-shell, perl, python, awk, or tcl/tk for daily tasks ranging from driving synthesis tools and embedding graphical user interfaces (GUI) to parsing large text files for report generation and file format conversion. The Internet also popularized scripting with CGI programming on the server side as well as scripting on the browser side. The emergence of MatLab in not only education but also hardware synthesis is further evidence that higher level languages with salient features of scripting are receiving wider attention. It has been shown that scripts are on average 10 times shorter than the corresponding programs in C, Java, or other system programming languages [1]. This translates into fewer bugs, higher productivity, and programs that are easier to understand.

However, scripting is often out of consideration for embedded systems programming for several reasons. First, most scripts are interpreted interactively or in batch, and many programmers assume that the runtime interpretation overhead may be prohibitive. Second, most interpreters built for general-purpose computers are written as application programs running on top of a general-purpose operating system with support for many features, and the memory and performance requirements tend to be too high for embedded systems. These assumptions have prevented embedded systems programmers from taking advantage of scripting. We show that not only can the interpretation overhead be minimized or eliminated, but scripting can actually encourage interactive testing at different levels of abstraction. By moving testing into an integral part of programming, we expect programmers to be able to develop and deploy working software in a much shorter amount of time.

Today, even though many of today's systems are becoming much more distributed and networked, software for these systems is still being developed in C or assembly code for a single-processor, self-contained system. Multiple steps are required, namely compiling, linking, loading the firmware into EEPROM or flash memory, and rebooting the system, before the code can be tested. This is not very convenient, and as a result, most programmers tend to write a large amount of code before testing, if at all. Another problem is code reachability. That is, even if a routine is part of a firmware program, it is not always invoked in a controlled manner; instead, the program needs to be written in a way that is coordinated with the program state and input sequence before a particular routine can be invoked and tested. Because reachability is an undecidable problem in general, many routines may go untested for years in such programs. Even if a single system is thoroughly tested in isolation, it is very difficult to test multiple systems working together in their target settings.

To address these problems, we propose a new approach based on a tool called Rappit [2]. Rappit is a design tool that tracks the configurations of a collection of networked, distributed embedded systems (henceforth called "nodes") and synthesizes light-weight scripting engines for them. The scripting engine consists

of a simple parser that invokes the desired routines, a message generator, and the required native routines. Rappit also serves as a host-side environment for general programming (e.g., GUI, network, database) and for accessing a single node or a group of nodes. The same scripting mechanism is used both during development and after deployment.

Our Rappit approach brings to embedded systems many benefits of interactive scripting. Scripting enables black-box reuse of software primitives, which may be arbitrarily simple or arbitrarily complex. The user can interactively type in these commands and get immediate feedback without having to go through the compile-link-load-reboot cycle. These commands may be very low level and architecture specific, such as setting an I/O pin, configuring a timer, or reading a status register. They may also be very high level, such as an FFT operation or processing an entire image. The interactivity encourages programmers to test every single line they write, and the instant validation enables them to write correct, working code quickly. The functionally correct script code can then be either interpreted or translated and compiled into native code by Rappit for efficiency or for raising the level of abstraction. The scripting-engine synthesis feature of Rappit also minimizes the interpretation overhead. In fact, since most distributed embedded systems must be able to handle messages anyway, scripting can be viewed as a generalized message handling mechanism. The incremental overhead above the an ad hoc message handler can be very small.

In addition to facilitating testing of a single node, scripting with Rappit also makes possible testing a network of nodes. On the host side, Rappit treats each node as a data object in its scripting environment. It is easy to send the same script or different scripts and test cases to these nodes by generating them programmatically. Moreover, the Rappit host-side environment can run a simulation program to emulate a node that interacts with real nodes by exchanging scripts with them. This will enable in-field testing by considering not only the functionality but also actual timing and environmental effects such as RF interference. This is particularly useful for wireless sensor applications, where the sensory input may be difficult to reproduce.

Although the productivity gain is difficult to quantify, in our experience, our scripting approach has dramatically reduced development time and effort in a number of projects. For system programming, scripting made it possible to write and test a device driver from scratch for a custom digital radio transceiver in a matter of hours instead of days or even weeks. For application programming, scripting enabled programmers to experiment with API calls and confirm the behavior instantly, and this has shown similar accelerated development time. In both cases, we believe that by eliminating the steps required between programming and testing by providing programmers with an interactive scriptable environment, programmers will naturally want to test their code as often as possible. With proper support from a tool for code synthesis, embedded systems programmers may just be able to reap the full benefit of scripting for the first time.

References

[1] John K. Ousterhout. Scripting: Higher level programming for the 21st century. *IEEE Computer*, 31(3):23–30, March 1998.

[2] Jiwon Hahn, Qiang Xie, and Pai H. Chou. Rappit: A framework for the synthesis of host-assisted light-weight scripting engines for adaptive embedded systems. In *Proc. International Conference on Hardware Software Codesign and System Synthesis (CODES+ISSS)*, pages 315–320, September 2005.