Chapter 10
Engineering Embedded Software
From Application Modeling to Software Synthesis

Ronaldo Ferreira
Universidade Federal do Rio Grande do Sul (UFRGS), Brazil

Lisane Brisolara
Universidade Federal de Pelotas (UFPEL), Brazil

Julio C.B. Mattos
Universidade Federal de Pelotas (UFPEL), Brazil

Emilena Spech
Universidade Federal do Rio Grande do Sul (UFRGS), Brazil

Erika Cota
Universidade Federal do Rio Grande do Sul (UFRGS), Brazil

Luigi Carro
Universidade Federal do Rio Grande do Sul (UFRGS), Brazil

ABSTRACT

Since 1965, with Moore's law statement, industry is continually aggregating complex services into their products, enhancing people's life quality with decreasing prices. Despite the advances towards hardware integration, current electronic products are relying even more on software to offer distinguished functionalities to users. Hence, the embedded system industry is facing a paradigm shift from its old fashioned hardware driven development to a strong software based one, exposing to the embedded systems domain unforeseen software design challenges. Indeed, this domain must devise its own and very specialized software engineering techniques, in order to achieve sustainable market growth with quality in the scheduled time. Embedded software is distinct from the standard one, fundamentally in the sense that its development is driven by physical properties such as memory footprint and energy consumption. Furthermore, embedded systems are developed within a very tight time-to-market window, pushing design and development practices to their limit. In this chapter, we discuss the use of software specifications at higher abstraction levels and the need to provide tools for software automation, because reliability

DOI: 10.4018/978-1-60566-750-8.ch010

and safety are important criteria present in several embedded applications, as well as time-to-market. This chapter discusses the design flow for embedded software, from its modeling to its deployment in the embedded platform.

INTRODUCTION

For a long time now, personal and server computers amount to less than 2% of the processor market, where standard software executes on (Turley, 2002). Most of the developed software runs on the remaining 98% of the processors available in the market. The embedded system domain is driven by reliability, cost, and time-to-market factors (Graaf, Lormans, & Toetenel, 2003). It usually has hard constraints regarding performance, memory, and power consumption, among other physical properties. Embedded systems are also often used in life-critical situations, where reliability and safety are more important criteria than pure performance (Edwards, Lavagno, Lee, & Sangiovanni-Vincentelli, 1997; Koopman, 2007).

Embedded software has always been associated to a low-level, relatively simple code responsible for a few specific tasks, such as hardware configuration and device drivers. However, such a view is no longer a reality. In currently developed applications, software is advancing to constitute the main part of the system, and in many cases software solutions are preferred to hardware ones, because software is more flexible, easier to update, and can be reused (Graaf, Lormans, & Toetenel, 2003). In addition, industry statistics from (Venture Development Corporation, 2006) reveal that the amount of lines of embedded code is growing at about 26% yearly. Although this data cannot be considered alone, it is an important indicative of software complexity (Kan, 2002). Due to this growing complexity and short time-to-market window, statistics from (CMP Media, 2006) show that more than one half of current embedded design projects are running behind schedule. This delay is mostly caused by the software productivity and testing gap, where hardware is almost being considered a commodity product, since the introduction of platform-based design (Sangiovanni-Vincentelli & Martin, 2001).

To cope with the increasing complexity, developers have been looking for higher levels of abstraction during system specification. Abstraction makes the development easier, and the final program becomes more readable and less hard to maintain. This fact is known by software engineers since the 1960's, when the software crisis was first discussed in the NATO conference (Naur & Randell, 1969), and important advances on the software development methods and techniques have been proposed since then. Such advances help tackle the complexity of a million-line code with the use of software languages, management approaches, testing tools, and modeling methods (Osterweil, 2007).

There are several well-established software development processes, such as the Unified Process and Agile Development Methods (Cockburn, 2001), whose usage is now supported by the standard system modeling language UML (OMG, 1997) and by CASE (Computer Aided Software Engineering) tools (Pressman, 2004). Today, one can think of *software product lines*, producing code and delivering software services in a rate that could never be imagined in 1968 during the NATO conference. Furthermore, software companies may now rely on certifications over the development process as a way to achieve software quality. This is the goal, for instance, of the Capability Maturity Model (CMM) and the Capability Maturity Model Integration (CMMI)