Preface

Embedded systems development requires integration of a variety of hardware and software components. This field, due to its multifaceted nature, has received contributions from different disciplines, namely computer science, computer engineering, software engineering, hardware-software codesign, and system modeling, not to mention mechanical engineering and materials sciences and engineering.

Model-driven development techniques are gaining importance, especially in the software community, and are expected to become a mainstream engineering approach in the near future, due to the benefits that they offer. Model-driven approaches offer a higher degree of abstraction, which is a crucial characteristic to tackle the growing complexity of the modern computer-based systems. This implies that in the next couple of years the application of model-driven approaches to develop embedded systems is likely to be not only useful, but a real necessity to address the growing complexity of those systems and fight against what is commonly called as the "productivity gap" (as far as methods and tools support are evolving at a small pace than chip complexity does, according with Moore's law). Therefore, system developers are supposed to master modeling techniques to perform their development tasks.

This book is about embedded computer systems. This book is also about how to model the behavior of those systems, for design and implementation purposes. Its technical contents focus on providing a state-of-the-art overview on behavioral models currently used for developing embedded systems, emphasizing on graphical and visual notations. You can find an excellent set of chapters, written in a tutorial-like style by leading and respected researchers and industry practitioners in the field of behavioral modeling of embedded systems. The chapters of this book cover a wide range of the important topics related to modeling embedded systems and were carefully selected and reviewed to give a coherent and broad view of the field.

CHARACTERISTICS OF EMBEDDED SYSTEMS

Defining succinctly and precisely what is an embedded computing system seems literally impossible, since embedded devices are very diverse and encompass a big variety of different applications. As Koopman, et al. assert, "*embedded computing is more readily defined by what it is not (it is not generic application software executing on the main CPU of a "desktop computer") than what it is*" (1996). Thus, personal computers are not considered to be embedded systems, even though they are often used to build embedded systems (Wolf, 2001).

An embedded system can be defined as a combination of computer hardware and software, and perhaps additional mechanical and others parts, developed to perform a specific or dedicated function.

Embedded systems are computer-based systems conceived for a specific application, using software and hardware components (Kumar, et al., 1996).

Generally, an embedded system is part, as its designation suggests, of a more complex system in which it is physically incorporated. The term "embedded," which was popularized by the U.S. Department of Defense, refers to the fact that those systems are included in bigger systems, whose main function is not computation. Therefore, a system can be understood as embedded, if its observer foresees its use within the context of a broader system or environment.

Embedded systems are typically built to control a given physical environment, by directly interacting with electrical and electronic devices and indirectly with mechanical equipments. For this purpose, an embedded system uses a set of sensors and actuators that allow, respectively, to receive information from and to send commands to the surrounding environment.

The class of embedded systems represents a very large percentage of the systems we daily use, either at home and at industrial facilities. Typical examples of embedded systems are control systems for cars, home appliances, printers or industrial machines; systems to acquire data from laboratorial equipments; systems to support medical diagnosis; or systems to control or supervise industrial processes.

The most relevant characteristics of embedded systems are the following (Camposano and Wilberg, 1996):

- An embedded system is usually developed to support a specific function for a given application. In some cases, only a single system is put into work; in others, the system is supposed to be a mass-market product.
- Embedded systems are expected to work continuously, that is they must operate while the bigger system is also operating.
- An embedded system should keep a permanent interaction with its surrounding respective. This means that it must continuously respond to different events coming from the environment, whose order and timing of occurrence are generally unpredictable.
- Embedded system need to be correctly specified and developed, since they typically accomplish tasks that are critical, both in terms of reliability and safety. A unique error may represent severe losses, either financial or, even worse, in terms of human lives.
- Sometimes, embedded systems must obey some temporal restrictions. Thus, real-time issues must be dealt with. As pointed out by Zave: "*Embedded is almost synonymous with real-time*" (1982).
- Nowadays, almost all computer-based systems are digital, and embedded systems also follow this trend (at least at their core).

CATEGORIES OF EMBEDDED SYSTEMS

Embedded applications vary so much in characteristics: two embedded systems may be so different, that any resemblance between both of them is hardly noticed. In fact, the term "embedded" covers a surprisingly diverse spectrum of systems and applications, including simple control systems, such as the controller of a washing machine implemented in a 4-bit microcontroller, but also complex multimedia or telecommunication devices with severe real-time constraints or distributed industrial shop-floor controllers.

Embedded systems can be found in applications with varying requirements and constraints such as (Grimheden, et al., 2005):

- Different production units, from a unique system for a specific client to mass market series, implying distinct cost constraints.
- Different types of requirements, from informal to very stringent ones, possibly combining quality issues, such as safety, reliability, real-time, and flexibility.
- Different operations, from short-life to permanent operation.
- Different environmental conditions in terms of radiation, vibrations, and humidity.
- Different application characteristics resulting in static versus dynamic loads, slow to fast speed, compute vs. interface intensive tasks.
- Different models of computation, from discrete-event models to continuous time dynamics.

Koopman, et al. divide, from an application point of view, embedded computing in the following twelve distinct categories (2005): (1) small and single microcontroller applications, (2) control systems, (3) distributed embedded control, (4) system on chip, (5) networking, (6) embedded PCs, (7) critical systems, (8) robotics, (9) computer peripherals, (10) wireless data systems, (11) signal processing, and (12) command and control.

This large number of application areas, with very distinct characteristics, leads us to the inevitable conclusion that embedded systems are quite hard to define and describe. This clearly makes generalizations difficult and complicates the treatment of embedded systems as a field of engineering.

One possible solution is to agree on a division of the embedded field, and consider, for example, four main categories: (1) signal-processing systems, (2) mission critical control systems, (3) distributed control systems, and (4) small consumer electronic devices (Koopman, et al., 1996). For each category, different attributes, such as computing speed, I/O transfer rates, memory size, and development costs apply. Furthermore, distinct models of computation, design patterns and modeling styles are also associated with those categories.

An alternative and simpler classification is proposed in (Edwards, et al., 1997): (1) reactive, (2) interactive, and (3) transformational embedded systems. The important message to retain here is that generalization about embedded systems may sometimes only apply to a specific category.

TYPES OF EMBEDDED SYSTEMS

Even though some computing scientists consider very naïvely or arrogantly that embedded software is just software that is executed by small computers, the design of this kind of software seems to be tremendously difficult (Wirth, 1997). An evident example of this is the fact that PDAs must support devices, operating systems, and user applications, just as PCs do, but with more severe cost and power constraints (Wolf, 2002). In this section, we argue that embedded software is so diverse from conventional desktop software that new paradigms of computation, methods and techniques, specifically devised for developing it, need to be devised and taught.

The principal role of embedded software is not the transformation of data, but rather the interaction with the physical world, which is apparently the main source of complexity in real-time and embedded software (Selic, 1999). The role of embedded software is to configure the computer platform in order to

meet the physical requirements. Software that interacts with the physical environment, through sensors and actuators, must acquire some properties of the physical world; it takes time to execute, it consumes power, and it does not terminate (unless it fails). This clearly and largely contrasts with the classical notion of software as the realization of mathematical functions as procedures, which map inputs into outputs. In traditional software, the logical correctness of the algorithm is the principal requirement, but this is not sufficient for embedded software (Sztipanovits & Karsai, 2001).

Another major difference is that embedded software is developed to be run on machines that are not just computers, but rather on cars, radars, airplanes, telephones, audio equipments, mobile phones, instruments, robots, digital cameras, toys, security systems, medical devices, network routers, elevators, television sets, printers, scanners, climate control systems, industrial systems, and so on. An embedded system can be defined as an electronic system that uses computers to accomplish some specific task, without being explicitly distinguished as a computer device. The term "embedded", coined by the U.S. DoD, comes actually from this characteristic, meaning that it is included in a bigger system whose main (the ultimate) function is not computation. This classification scheme excludes, for example, desktop and laptop computers from being embedded systems, since these machines are constructed to explicitly support general-purpose computing. As a consequence of those divergent characteristics, the embedded processors are also quite different from the desktop processors (Conte, 2002).

The behavior of embedded systems is typically restricted by time, even though they may not necessarily have real-time constraints (Stankovic, 1996). The correctness of a real-time system depends not only on the logical results of the computation, but also on the time at which those results are produced (Stankovic, 1998). A common misconception is that a real-time system must respond in microseconds, which implies the need to program it in a low-level assembly language. Although some real-time systems do require this type of answer, this is not at all universal. For example, a system for predicting the weather for the next day is a real-time system, since it must give an answer before the day being forecasted starts, but not necessarily in the next second; if this restriction is not fulfilled, the prediction, even if correct, is useless from a practical point of view.

A *real-time system* is a system whose behavior must respect the intended functionality but also a set of temporal restrictions, externally defined (Benveniste & Berry, 1991). A critical aspect or a real-time system, as it name clearly suggests, is the way temporal issues are handled. When developing a system of this type, the temporal requirements must be identified and the development team must make sure that they will be fulfilled when the system is in operation.

It is common to classify real-time systems in terms of the reaction time that they exhibit with respect to the needs of the environment where they are located. Thus, the temporal restrictions that affect the systems can be divided into three main categories, each one corresponding to a different type of real-time systems:

- **Hard real-time systems:** An answer to be correct needs to take into account also the instant when it occurs. A late answer is incorrect and constitutes a failure of the system. For example, if a moving robot takes more time than required to decide what to do when avoiding an obstacle a disaster may occur.
- **Soft real-time systems:** The requirements in this category are specified with a medium value for the answer. If an answer comes late, the system does not fail; instead the performance of the system degrades. For example, if a file takes too long to open (i.e. takes more time than expected), the user might become dissatisfied (or even desperate), but no failure results from this fact.

- **Firm real-time systems:** This represents a compromise between the two previous categories, where a low probability is tolerated in the non-fulfillment of a response time that conducts to a failure of the system, mostly affecting associated quality of service.

Embedded systems are also typically influenced in their development by other constraints, rather than just time-related ones. Among them one can include: liveness, reactivity, heterogeneity, reliability, and distribution. All these features are essential to guarantee the correctness of an embedded program. In particular, embedded systems are strongly influenced in their design by the characteristics of the underlying computing platform, which includes the computing hardware, an operating system, and eventually a programming framework (such as .NET or EJB) (Selic, 2002). Thus, designing embedded software without taking into account the hardware requirements is nearly impossible, which implies that, at least currently, "write once run anywhere" (WORA) and the model-driven architecture (MDA) principles are not easily or directly applicable.

Reactive systems, a class of systems in which embedded systems can be included, maintain a permanent or frequent interaction with their respective environment and respond based on their internal state to the external stimulus to which they are subject to. Reactive systems cannot be described by specifying the output signals in function of the input signals; instead, specifications that relate inputs and outputs along the time are required. Typically, descriptions of reactive systems include sequences of events, actions, conditions and flows of information, combined with temporal restrictions, to define the global behavior of the system.

Using terminology borrowed from the classical discipline of digital systems, a reactive system is classified as a sequential system (and not as a combinational system). A *combinational logic system* is described by logical functions, has a set of inputs and outputs, and the values of the latter depend exclusively on the values of the inputs and on the internal constitution of the system. In a *sequential logic system*, the output values, in a given moment, do not depend exclusively on the input values at that moment, but also on the past sequences of the values that were present in the inputs (Wakerly, 2000). It may happen that, for equal combinations of the inputs, different values in the outputs are obtained, in different points in time. Sequential systems contain the notion of memory, contrarily to what happens with combinational systems.

Reactive systems have concurrency as their essential feature (Manna & Pnueli, 2002). Put in other words, development of embedded software requires models of computation that explicitly support concurrency. Although software must not be, at all, executed in sequence, it is almost universally taken for granted that it will run on a von Neumann architecture and thus, in practice, it is conceived as a sequential process. Since concurrency is inherent in all embedded systems, it must be undoubtedly included in every development effort.

With all these important distinctions, one clearly notices that the approaches and methods that are used for traditional software are not suitable for embedded software. Embedded software requires different methods, techniques, and models from those used generically for software. The methods used for non-embedded software require, at a minimum, major modifications for embedded software; at a maximum, entirely new abstractions are needed that support physical aspects and ensure robustness (Lee, 2002).

The inadequacy of the traditional methods of software engineering for developing embedded systems appears to be caused by the increasing complexity of the software applications and their real-time and safety requirements (Balarin, et al., 2002). These authors claim that the sequential paradigm, embodied

in several programming languages, some object-oriented ones included, is not satisfactory to adequately model embedded software, since this type of software is inherently concurrent.

One of the problems of object-oriented design, in what concerns its applicability for embedded software, is that it emphasizes inheritance and procedural interfaces. Object-oriented methods are good at analyzing and designing information-intensive applications, but are less efficient, sometimes even inadequate, for a large class of embedded systems, namely those that utilize complex architectures to achieve high-performance (Bhatt & Shackleton, 1998).

According to Lee, for embedded software, we need a different approach that allows us to build complex systems by assembling components, but whose focus is concurrency and communication abstractions, and admits time as a major concept (Lee, 2002). He suggests the term "actor-oriented design" for a refactored software architecture, where the components are not objects, but instead are parameterized actors with ports.

Actors provide a uniform abstract representation of concurrent and distributed systems and improve on the sequential limitations of passive objects, allowing them to carry out computation in a concurrent way (Hewitt, 1977; Agha, 1986). Each actor is asynchronous and carries out its activities potentially in parallel with other actors, being thus control distributed among different actors (Rens & Agha, 1998). The ports and the parameters define the interface of an actor. A port represents an interaction with other actors, but does not necessarily have call-return semantics. Its precise semantics depends on the model of computation, but conceptually it just represents communication between components.

BEHAVIORAL MODELING

Abstraction and modeling are two related techniques for reasoning about systems. Abstraction is a process that selectively removes some information from a description to focus on the information that remains. It is an essential part of modeling where some aspect or part of the real world is simplified. Thus, abstraction is related to the notion of simplicity.

Modeling is the cost-effective use of something in place of something else for a given purpose. A model represents reality for the considered purpose and allows engineers and developers to use something that is simpler, safer or cheaper than reality. A model is an abstraction of reality, since it cannot represent all its aspects. Therefore, the world is dealt in a simplified manner, avoiding the complexity, the danger and the irreversibility of reality.

To be useful and effective, a model of a given system must possess the following five key characteristics (Selic, 2003):

- **Abstract:** The model must concentrates on the crucial aspects of reality for the purpose at hand. The aspects that are not important in a given situation must be neglected or hidden.
- **Understandable:** The reduced form of the model, when compared with reality, must still be sufficiently expressive to allow us to reason about the properties of the system.
- **Accurate:** The key features of the system under consideration must be represented by the model.
- **Predictive:** The model must permit the system's properties to be predicted, either by experimentation or by formal analysis.

- **Inexpensive:** The model must be significantly cheaper to build and analyzed that the system that it represents.

According to Sgroi, et al. (2000), usage of behavioral modeling in embedded system design may contribute:

- To unambiguously capture the required system's functionality.
- To verify the specification correctness against intended proprieties.
- To support synthesis into specific platforms.
- To support usage of different tools based on the same model, allowing interoperability and coverability of different development phases, including designing, producing, and maintaining the system.

In the previous paragraphs, some of the major advantages of model-based development were highlighted. Availability of a model for the system promotes a better understanding of the system, even before starting its design, and allowing the comparison of different approaches, forcing desired proprieties to be present, and detect and remove undesired proprieties in advance. This model-based development attitude is supported by adequate modeling formalisms.

A multitude of modeling formalisms have been proposed for embedded systems design emphasizing their behavior description, ranging from those formalisms heavily relying on a graphical representation to those supported by textual languages. Another dichotomy for classification of modeling formalisms relies on control versus data dominated formalisms. Another possible classification is based on emphasizing specific aspects, like the reactive nature of the system's behavior, the real-time response constraint, or the data processing capabilities.

Independently of the way we prefer to classify modeling formalisms, one crucial decision that the designer of embedded systems needs to take is the selection of the "right" formalism to use. This decision potentially constrains all the development steps and imposes the level of sophistication of the tools that support the process.

Some modeling formalisms for embedded systems are control-dominated, where data processing characteristics and computation are minimal, emphasizing the reactive nature of the system's behavior. Other modeling formalisms emphasize the data processing characteristic, integrating complex data transformations, and are normally described by data flows. For example, reactive control systems are clearly in the first group, while digital signal processing applications are clearly in the second, as they emphasize the usage of data flow models.

Additionally, it is also common to build distributed embedded systems, where one needs to face heterogeneity in terms of implementation platforms, as the different components of the system need to be mapped into different platforms due to cost and performance issues. In these situations, it is not easy to find a unique formalism to model the whole system, so the goal is to decompose the system into components, to model them using submodels and pick up the right formalism for the different submodels. In this situation, communication and concurrency enter into the picture and verification of the properties of the whole system could become more difficult to handle. It has to be stressed that model-based development adequately supports verification of properties, which is a subject of major importance in embedded system design, especially when the system complexity is high.

BOOK CONTENTS

This book is composed of 15 chapters that are divided into five main sections.

The first section is about model-based approaches to support some of the development activities associated with embedded systems. Due to the huge popularity of the unified modeling language (UML), model-based approaches are nowadays available in almost all software and hardware engineering branches, and the embedded field is obviously no exception. Brisolara, Kreutz, and Carro present in **Chapter 1** the use of the UML as a modeling language for designing embedded systems. The authors also discuss the need to extend the UML through profiles in order to address the specific mechanisms associated with embedded computing. In **Chapter 2**, Gargantini Riccobene, and Scandurra deal with a model-driven approach based on UML and on abstract state machines (ASM) for supporting the design process system-on-chip (SoC). The approach supports also the generation of SystemC code, thus facilitating and speeding the implementation phase. **Chapter 3**, written by Baloukas and other colleagues, discusses how to optimize data types for dynamic embedded systems. The chapter describes UML transformations at the modeling level and C/C++ transformations at the software implementation level. These two types of transformations focus on the data types of dynamic embedded software applications and provide optimizations guided by the relevant cost factors (e.g., memory footprint and the number of memory accesses).

The second section includes three chapters on approaches that use aspect-oriented concepts, namely the separation of concerns, to address the modeling of embedded systems. Gray and others present in **Chapter 4** a model-driven approach for generating quality-of-service (QoS) adaptation rules in distributed real-time embedded systems. Their approach creates graphical models representing QoS adaptation policies, which are expressed with the adaptive quality modeling language (AQML), a domain-specific modeling language that helps in separating common concerns of an embedded system via different views. **Chapter 5**, written by researchers from UFRGS, Brazil, introduces an approach, based on concepts from model-driven engineering (MDE) and aspect-oriented design (AOD), for exploring the design space of embedded systems. The authors show how their approach achieves better reusability, complexity management, and design automation by exploiting simultaneously ideas and principles from MDE and AOD. **Chapter 6** introduces a model-based framework for embedded real-time systems, proposed by de Niz, Bhatia, and Rajkumar, all affiliated with CMU. Their approach enables a decomposition structure that reduces the complexity of both functional and parafunctional aspects of the software. This decomposition enables the separation of the functional and parafunctional aspects of the system into semantic dimensions (e.g., event-flow, timing, deployment, fault-tolerant) that can be represented, manipulated, and modified independently of the others from an end-user perspective.

The third section tackles issues related with verification, a development activity that focuses on formally proving that the system implementation is in accordance with the respective specification. In particular, model-checking techniques for real-time embedded systems are addressed. **Chapter 7**, authored by Rodriguez-Navas, Proenza, Hansson, and Pettersson, is devoted to modeling clocks in distributed embedded systems using the timed automata formalism. The discussion is centered around the UPPAAL model checker, which is based on the theory of timed automata. The different computer clocks that may be used in a distributed embedded system and their effects on the temporal behavior of the system are discussed, together with a systematic presentation of how the behavior of each type of clock can be modeled. In **Chapter 8**, Waszniowski and Hanzálek from the Czech Technical University show how a multitasking real-time application, consisting of several preemptive tasks and interrupt service routines

potentially synchronized by events and sharing resources, can be modeled by timed automata. Again, the UPPAAL model checker is used, but in this case to verify the timing aspects and the logical properties of the proposed model. In **Chapter 9**, Posadas and his colleagues focus on a framework based on SystemC for platform modeling, software behavioral simulation, and performance estimation of embedded systems. With the framework, the application software running on the different processors of the platform can be simulated efficiently in close interaction with the rest of the platform components. Therefore, design-space exploration can be supported by fast and sufficiently accurate performance metrics.

The fourth section concentrates on several topics related with automating the development process (also named design process or design flow). **Chapter 10**, written by Ferreira and other coauthors, discusses the use of software specifications at higher abstraction levels and the need to provide tools for software automation, because quality issues, like reliability, safety, and time-to-market, are important aspects that have a great impact in the development of many embedded applications. This chapter discusses the complete design flow for embedded software, from its modeling to its deployment. In **Chapter 11**, Yu, Abdi, and Gajski address automation issues for multicore systems based on well-defined transaction level model (TLM) semantics. TLMs replace the traditional signal toggling model of system communication with function calls, thereby increasing simulation speed. TLMs play a pivotal role in the development activities before the final prototype is implemented. The authors discuss important topics in TLM automation and also provide an understanding of the basic building blocks of TLMs. Cardoso and his coauthors provide in **Chapter 12** an overview of reconfigurable computing concepts and programming paradigms for the reconfigurable computing architectures. Two major aspects are taken into consideration: (1) how the programming model can aid on the mapping of computations to these architectures, and (2) how the programming models can be used to develop applications to these architectures.

The fifth and last section includes three chapters that present and discuss several issues that are relevant in real industrial contexts, especially in the automotive domain. Khalgui and Hanisch discuss in **Chapter 13** reconfiguration issues in the context of industrial control applications and in particular the development of safety reconfigurable embedded control systems following the international industrial standard IEC61499. The authors define a new semantic of the reconfiguration, by considering the improvement of the system performance at run-time as a major criterion. **Chapter 14**, written by Faucou, Simonot-Lion, and Trinquet, deals with the EAST-ADL, an architecture description language dedicated to the automotive domain, with a focus on the support provided concerning the validation and verification activities. Finally, in **Chapter 15**, Zander-Nowicka and Schieferdecker introduce model-based testing methods for embedded systems in the context of the automotive domain.

Luis Gomes and João M. Fernandes
Lisbon and Braga, Portugal
January 2009

REFERENCES

Agha, G. (1986). *Actors: A model of concurrent computation in distributed systems*. MIT Press.

Balarin, F., Lavagno, L., Passerone, C., & Watanabe, Y. (2002). Processes, interfaces, and platforms. Embedded software modeling in metropolis. *2nd International Workshop on Embedded Software (EMSOFT 2002)* (pp. 407-421). Springer.

Benveniste, A., & Berry, G. (1991). The synchronous approach to reactive and real time systems. In *Proceedings of the IEEE*, *79*(9), 1270-1282.

Bhatt, D., & Shackleton, J. (1998). A design notation and toolset for high-performance embedded systems development. Lectures on embedded systems. *European Educational Forum School on Embedded Systems* (pp. 249–267). Springer.

Camposano, R. & Wilberg, J. (1996). Embedded system design. *Design Automation for Embedded Systems, 1*(1/2), 5–50.

Conte, T. M. (2002). Choosing the brain(s) of an embedded system. *IEEE Computer 35*(7),106–107.

Edwards, S., Lavagno, L., Lee, E. A., & Sangiovanni-Vincentelli, A. (1997). Design of embedded systems: Formal models, validation, and synthesis. In *Proceedings of the IEEE, 85*(3),366–390.

Grimheden, M., & Törngren, M. (2005). What is embedded systems and how should it be taught?—results from a didactic analysis. *ACM Transactions on Embedded Computing Systems, 4*(3),633-651.

Hewitt, C. (1977). Viewing control structures as patterns of passing messages. *Journal of Artificial Intelligence, 8*(3), 323-364.

Koopman, P. (1996). Embedded system design issues (the rest of the story). *IEEE International Conference on Computer Design (ICCD '96)* (pp. 310–317).

Koopman, P., Choset, H., Gandhi, R., Krogh, B., Marculescu, D., Narasimhan, P., Paul, J. M., Rajkumar, R., Siewiorek, D., Smailagic, A., Steenkiste, P., Thomas, D. E., & Wang, C. (2005). Undergraduate embedded system education at Carnegie Mellon. *ACM Transactions on Embedded Computing Systems, 4*(3), 500-528.

Kumar, S., Aylor, J. H., Johnson, B. W., & Wulf, W. A. (1996). *The codesign of embedded systems: A unified hardware/software representation*. Kluwer Academic Publishers.

Lee, E. A. (2002). Embedded software. *Advances in Computers, 56*.

Manna, Z., & Pnueli, A. (1992). *The temporal logic of reactive and concurrent systems: Specification*. Springer.

Ren, S., & Agha, G. (1998). A modular approach for programming embedded systems. Lectures on embedded systems. *European Educational Forum School on Embedded Systems* (pp. 170-207). Springer.

Selic, B. (1999). Turning clockwise: Using UML in the real-time domain. *Communications of the ACM, 42*(10), 46-54, 1999.

Selic, B. (2002). Physical programming: Beyond mere logic. *Embedded Software, 2nd International Workshop on Embedded Software (EMSOFT 2002)* (pp. 399-406). Springer.

Selic, B. (2003). The pragmatics of model-driven development. *IEEE Software, 20*(5),19-25.

Sgroi, M., Lavagno, L., & Sangiovanni-Vincentelli, A. (2000). Formal models for embedded systems design. *IEEE Design and Test of Computers, 17*(2), 14-17.

Stankovic, J. A. (1996). Real-time and embedded systems. *ACM Computing Surveys, 28*(1), 205–8.

Stankovic, J. A. (1988). Misconceptions about real-time computing: A serious problem for next-generation systems. *IEEE Computer, 21*(10), 10-19.

Sztipanovits, J., & Karsai, G. (2001). Embedded software: Challenges and opportunities. 1st International Workshop on Embedded Software (EMSOFT 2001) (pp. 403-415). Springer.

Wakerly, J. F. (2000). *Digital design: Principles and practices, 3rd ed*. Prentice-Hall International.

Wirth, N. (2001). Embedded systems and real-time programming. *1st International Workshop on Embedded Software (EMSOFT 2001)* (pp. 486-492). Springer.

Wolf, W. (2001). *Computers as components: Principles of embedded computing system design*. San Francisco: Morgan Kaufmann.

Wolf, W. (2002). What is embedded computing? *IEEE Computer, 35*(1), 136-137.

Zave, P. (1982). An operational approach to requirements specification for embedded systems. *IEEE Transactions on Software Engineering, SE-8*(3), 250-269.