Guidelines for a Graduate Curriculum on Embedded Software and Systems

P. CASPI, A. SANGIOVANNI-VINCENTELLI, L. ALMEIDA, A. BENVENISTE, B. BOUYSSOUNOUSE, G. BUTTAZZO, I. CRNKOVIC, W. DAMM, J. ENGBLOM, G. FOLHER, M. GARCIA-VALLS, H. KOPETZ, Y. LAKHNECH, F. LAROUSSINIE, L. LAVAGNO, G. LIPARI, F. MARANINCHI, PH. PETI, J. DE LA PUENTE, N. SCAIFE, J. SIFAKIS, R. DE SIMONE, M. TORNGREN, P. VERÍSSIMO, A. J. WELLINGS, R. WILHELM, T. WILLEMSE, and W. YI
The Artist Education Group

The design of embedded real-time systems requires skills from multiple specific disciplines, including, but not limited to, control, computer science, and electronics. This often involves experts from differing backgrounds, who do not recognize that they address similar, if not identical, issues from complementary angles. Design methodologies are lacking in rigor and discipline so that demonstrating correctness of an embedded design, if at all possible, is a very expensive proposition that may delay significantly the introduction of a critical product. While the economic importance of embedded systems is widely acknowledged, academia has not paid enough attention to the education of a community of high-quality embedded system designers, an obvious difficulty being the need of interdisciplinarity in a period where specialization has been the target of most education systems. This paper presents the reflections that took place in the European Network of Excellence Artist leading us to propose principles and structured contents for building curricula on embedded software and systems.

Categories and Subject Descriptors: K.3.2 [**Computer and Information Science Education**]: *Accreditation, computer science education, curriculum, information systems education, literacy, self-assessment*; K.3 [**Computer and Education**]: K. Computing Milieux, The ACM Computing Classification System (1998), Overview of 1998 ACM Computing Classification System

General Terms: Computer Science Education, Curriculum, Information Systems Education

Additional Key Words and Phrases: Graduate curriculum, embedded systems, control, real-time, distributed systems, extrafunctional properties, architecture and design, labs

This work has been supported by the European Commission under grant IST-2001-34820.
This paper is a short version of the Artist deliverable W2.All.Y1 which is publicly available at www.artist-embedded.org/Education/Education.pdf.
The contents reported here are the result of the views and beliefs of the entire group of authors. The burden of putting manuscript together has mainly rested upon Paul Caspi and Alberto Sangiovanni-Vincentelli. Contributors outside Artist: N. Suri, R. Gupta, I. Lee, and L. Aceto agreed to publish their teaching documents. F. Dancoisne performed some proof-reading.
Author's address: P. Caspi, Verimag (CNRS), Centre Equation, F-38610 Gieres, France.

1. INTRODUCTION

Although computer-based embedded systems have been designed for more than 30 years in a variety of industrial domains, e.g., avionics and aerospace, railways, energy, and industrial control, their economic importance has grown exponentially as electronic components have made their way into everyday-use devices, such as cars, appliances, and mobile phones. The strategic dimension of embedded computing has not been appreciated until recently when a variety of initiatives in the United States and in Europe marked the beginning of a strong research and industrial effort aimed at improving design practices. In fact, it has become painfully clear that, as complexity and pervasiveness of these devices grow, the concerns on performance, reliability and durability increase exponentially thus provoking a productivity crisis of major proportions. As a response to these concerns, the European Commission launched in 2001 the Artist FP5 Accompanying Measure, gathering more than twenty top academic institutions and laboratories, to promote and improve European cooperation on the subject.

In this setting, education has indeed come to the forefront as an essential factor in creating a workforce that could cope effectively with the design challenges of future devices. Artist initiated a debate among its members to discuss the education issue. As an outcome of this debate, we proposed a work-package on education, with the goal of investigating whether current education approaches meet industrial and research needs and, in the case it does not, how to construct a novel curriculum in embedded systems. This paper summarizes the reflections that took place in the consortium. A few words of caution: while we are aware that embedded system design involves both hardware and software expertise, in this paper we focused mostly on its software aspects to limit its scope and provide enough depth of treatment of the subject. In general we will assume basic knowledge of hardware design, that is part of basic CS, CE and EE curricula, and will add some hardware-oriented topics whenever they are essential to apply the theoretical knowledge and complement the software aspects.

Given the scope of the project, the Artist Education Group took substantial amount of time in analyzing the education landscape and in examining the problems it raises. The reflections that stemmed from this analysis, are presented in Section 2. From this analysis, we have drawn both the principles that should drive education in the domain of embedded systems and the limits of our possible contribution. Both are presented in Section 3. In particular, the question of looking at the entire education platform was considered to be too large of a goal and a narrower one was chosen, namely how to design an "ideal" graduate curriculum for embedded systems. We deliberately stayed away from addressing the actual implementation issues (e.g., organization, timing, and partition into different courses) of the ideal curriculum in a very diverse community such as the one formed by the countries in the EU: different nations impose their own rules and follow their own tradition in implementing curricula. Consequently, we focused on identifying the large bodies of knowledge that should in some form or another participate in these curricula. The resulting contents are presented in Section 4. Finally, we conclude by discussing how these reflections can be cast as an effective road map toward curriculum development.

Related Work. A number of approaches have been proposed for the definition of computer science and engineering curricula, in various academic, industrial or mixed contexts. Some of them are closely related to our subject. Others are more general. Among the various approaches available, we selected for their relevance: the IEEE/ACM guidelines, CareerSpace, the Swedish national network ARTES and the U.S. Center for Hybrid and Embedded Software Systems (CHESS) NSF ITR.

1. IEEE/ACM (www.acm.org/education/curricula.html) guidelines are very general, and do not seem to address one of the main aspects we focus on in this document: a curriculum on embedded systems has to borrow courses from various teachings contexts, such as computer science, control theory, dependable systems.
2. CareerSpace (www.career-space.com/) is focused on Information and Communications Technology (ICT). The list of *job profiles* they propose for the domain does not mention embedded systems. The part entitled "Recommendations for Designing New ICT Curricula" is worth reading, for the curriculum structure proposed.
3. The ARTES (www.artes.uu.se/) Swedish strategic research initiative in Real-Time Systems is a network of academic and industrial groups, with the ambition to strengthen the Real-Time Systems competence nationwide. The main focus of ARTES is on graduate education and cooperation between industry and academia. Concerning education, ARTES created a number of useful courses in real-time systems. Furthermore, ARTES has promoted the mobility of graduate students all over Sweden. This approach has helped increase the efficiency of graduate education, and the exchange of people and ideas.
4. CHESS (chess.eecs.berkeley.edu/) is funded in part by an Information Technology Research (ITR) project from the National Science Foundation (NSF). It operates cooperatively with the Institute for Software Integrated Systems (ISIS) at Vanderbilt University. This center is aimed at developing model-based and tool-supported design methodologies for real-time fault tolerant software on heterogeneous distributed platforms. The Center attempts to bridge the gap between computer science and systems science by developing the foundations of a modern systems science that is simultaneously computational and physical. This represents a major departure from the current, separated structure of computer science (CS), computer engineering (CE), and electrical engineering (EE): it reintegrates information and physical sciences. As part of its mission, education material and curricula are developed for courses in the embedded system domain (See the paper *An Overview of Embedded System Design Education at Berkeley* in this issue). An outreach program is an integral part of the ITR where this material is proactively distributed to community colleges in California and

minority colleges in the rest of the United States. The outreach program helps instructors to teach the material of graduate and undergraduate courses by hosting them during summer break in Berkeley.

2. THE EMBEDDED SYSTEM EDUCATION LANDSCAPE

The domain of embedded system engineering and science has emerged from a diverse cultural background: from mechanical engineering to electrical engineering, from communications to computers. Historically, the design of embedded systems was carried out with different methods according to the particular application domain. Only recently, has there been an attempt at unifying the discipline once it has become clear that fundamental problems are shared across different application domains. However, because of its relative novelty, the field still lacks maturity, so much so that teaching embedded software design from a unified perspective is a real challenge. In this section, we list some of the obstacles that need to be overcome to yield a robust teaching approach.

2.1 Diversity of Origins

Embedded systems engineering and science is certainly not a unified field. Historically, the design methods and approaches were particular to the implementation domain, thus resulting in a fragmented landscape of *ad hoc techniques*. Examples of these domains are telecommunications, mechanics, automotive, aeronautics, and consumer products. It is indeed very interesting to trace back the evolution of these domains, starting from the nineteenth century and the time of mechanical implementations, going through the use of electromechanical devices, analog systems, and arriving at our modern age of microcomputers and system-on-a-chip. It is also fascinating to observe how each domain has designed its own intellectual tools, as a result of this diverse evolution.

2.2 Diversity of Cultures

The difference in lineage of embedded system engineering has led quite naturally to a large diversity of cultures. This can be observed almost everywhere, the most relevant aspect being the different notion of “time.” Time has been a fascination for many physicists, mathematicians, and philosophers. The way of interpreting time strongly characterizes the differences in cultural milieus of embedded system.

For the continuous control engineer, time is, above all, continuous akin to the time axis of, say, mechanics. When one moves to computing, this continuous time is abstracted into discrete time through the various theories of periodic sampling. Quite surprisingly, another variety of control engineers, the discrete-event ones, have another notion of discrete time, which is quite similar but aperiodic. Finally, arising from computer science and telecommunication, there is another notion of time, the asynchronous one, which is based on partial orders. Each assumption on the nature of time yields different properties of the systems under consideration. A framework where these notions of time can be embedded and rigorously compared is an important step toward establishing a unifying discipline.

Because of this diversity in basic notions, it is not very surprising that communication problems occur between people who belong to these different cultures. Consequently, it is not an easy task to get them cooperating on a given design. This, in part, explains why people tend to be trained and specialized in a given application domain and why there is very little mobility between application domains. And why it makes sense to try to develop a curriculum where the important concepts are appropriately abstracted and compared.

2.3 Diversity of Practices

Embedded-system area offers a very large choice of possible implementations.

The choice between hardware and software is an important one, which runs through the entire design process. Even within software, a large choice of practices can be found, ranging from control-based designs—supported by the very popular Matlab/Simulink/Stateflow toolbox—to more computer-based object-oriented tools such as UML. Moreover, development in assembly language and C are still important.

A continuous debate among practitioners is about the synchronization policy to adopt: according to their application domains, some designers prefer disciplined periodic sampling (the so-called “Time-Triggered” approach) to restrict the space of designs and ease the verification problem, while some others stick to a more liberal asynchronous policy (“Event-Triggered” approach) that offers better performance, but for which it is more difficult to verify designs.

Even within the same culture, different schools promote competing technologies. For instance, within computer science, some people promote an integrated language approach (e.g., Java and Ada), while some others favor a clean separation between languages and (real-time) operating systems.

The point to make here is that the choice of one approach or another is today the result of cultural and industrial bias as opposed to a rigorous analysis of benefits and drawbacks. According to many Artist participants, this is indeed a clear symptom of the lack of maturity of the embedded system considered as a domain on its own and the root of the difficulty in teaching embedded system design as a unified discipline.

2.4 Diversity of Education

We argued that each of the application domains has its own culture and tradition. Also, each has developed its own computer engineering education intended to fulfill the (supposed) needs of its specific domain. Thus, we have seen specialized computer engineering curricula devoted to a particular speciality, such as railway computer engineering or aeronautics computer engineering.

Each of these application domains has only a limited view of computing and takes it more as a mere technique than as a science on its own. This state of affairs has, in our opinion, dramatic consequences in that it makes it very difficult to bridge the cultural and practical gaps that have been previously observed to yield a unified discipline.

Given this fragmentation, the current state of education in embedded systems and software can be rather hard to investigate, since there are so many different actors and departments involved. For this reason, the Artist Group on Education has made a particular effort to include people from disciplines that lie outside the core Artist network expertise. By doing so, we have a fairly complete view of the discipline but we cannot claim we have obtained a complete view of the landscape.

Continuing Education. The people who pursue research and teaching may be expected to stay up-to-date thanks to the very nature of their jobs in academia. Those who go to industry will have to be continuously educated in order to stay in phase with the technical evolution. Continuous education may be achieved in many ways: (1) guided, such as in-house training, university programs targeted to the industrial audience (in the United States, universities are very active in continuous education programs as they also provide a good funding source), and tool-vendor programs; and (2) spontaneous, such as attending seminars in the field.

At present each of these programs has its drawbacks. In particular, in-house training is likely to enforce cultural homogeneity and as such, may not foster communication among the various people that have to cooperate in an engineering project such as suppliers, clients, and, in some highly critical embedded systems, certification authorities. This also contributes to the continued domain fragmentation into so many subdisciplines.

Tool-vendor training is often suspected of bias as vendors are not likely to emphasize the limits of their tools, nor to account for the offerings of their competitors. Moreover, this kind of training concentrates on interfaces and use models rather than on the principles the tools are based on. This does not encourage a deep understanding of the tool capabilities and can even lead to misuse, defeating the very purpose of these programs.

A well-organized and university-driven continuous education program would certainly go a long way to support professionals in industry, but it is not nearly enough. We need to provide students, when they are still at University, with a basis that will help them understand what they will need to learn during the 30 or 40 years of their career.

However, at the same time, we should also not forget that their potential employers will greatly benefit if they are efficient as soon as they are hired. Thus, we have to provide them with sufficient practical training to be easily integrated in the industrial landscape. We have to balance a good and strong cultural basis with good practices. This is why designing curricula in such a fast-moving discipline like ours is so difficult.

Diversity of Education Styles. Besides the diversity of education arising from different application domains, there is a large diversity of education styles and organization among different countries and between different educational institutions, even within the same country. For example, in some countries, curricula are rigid, with few choices offered to students; in others, students can almost freely choose their courses under the guidance of a supervisor. In addition, two opposite approaches to teaching are the subject of heated debate: (1) deductive that privileges theory and introduce applications as a consequence, and (2) inductive that builds theory by distilling special cases and applications.

Levels of Education. Embedded systems in the recent past have been considered an application area where students need to learn tools of the trade rather that a field of basic knowledge. In Europe, the PhD degree is seen mostly as a path to an academic career. Indeed, in some countries, the number of PhDs that pursue an industrial career is very small. Hence, very few PhDs were trained in embedded systems. This situation is most unfortunate in two respects: first of all, there was no critical mass of highly educated people who could advance the state of the art of embedded systems; second, (and this is not only a problem in embedded systems, but in engineering and computer science at large) it is well documented in the United States that PhDs have a fundamental role in innovation in established companies and in the creation of new enterprises. However, there is a general trend in the European industry to pay more attention to PhDs as their role in innovation is increasingly more evident.

3. PRINCIPLES AND LIMITATIONS

The principles and limitations of our contribution in designing curricula are based on our observations of the education landscape in the embedded system domain, on the situation of the Artist consortium within this landscape, and on general views of what higher engineering should consist of.

3.1 Principles

Emphasize the Role of Basic Knowledge and Computer Science. We believe that a solid foundation in the fundamentals of engineering and computer science will allow students to benefit from the continuous training they will be exposed to throughout their professional life. As we have seen, the embedded system domain is subject to a large variety of cultures and practices. For instance, depending on the specific culture of an application domain, different implementation approaches can be used for the same computation model. In many cases, these application domains tend to ignore computer science and consider computing as a mere technique. However, we firmly believe that computing is not just a technique and that it has elaborated a rich corpus of theory that is likely to help in putting some order in the complex landscape of embedded system practices. Hence, we believe that computer science has a major role to play, because it can provide a sound and unifying view on the various computation models. Moreover, it has specific bodies of knowledge, such as formal verification, that are essential for the development of correct embedded systems.

Increase Awareness to Application Domains. Computer science will succeed in this unifying role only if it is able to meet the needs of the embedded system application disciplines. This requires understanding the specific characteristics and problems of these disciplines as the role of computer science is not always obvious and has its maximum impact on the formalization of problems and their solution via rigorous methods.

Laboratories and experimental techniques will allow students to become sensitized to the practical problems of embedded systems. This will also allow them to innovate and be ready to solve important problems as soon as they obtain their first job.

Promote the Comparison of Several Approaches. We live in a world where techniques are evolving faster every day. In this situation, the debate about what should be taught in the classrooms is heated. In particular, there is always the risk that the decision about what to teach is driven by the desire of staying close to research by addressing the most fashionable topics. This aspect may cause student education to suffer from isolation from the basics of the topic, as well as from other approaches pursued in different groups. In addition, the evolution is so fast that we are likely to always lag behind it; the people we train now will be already behind with respect to the technical evolution as soon as they get their university degree!

Although there already exist some excellent courses and curricula in the domain of embedded systems (see the other papers in this issue), they tend to present a single approach. We believe that University is the right place where different approaches can be objectively compared and assessed. Industry is always under pressure to bring new products to market in time and, thus, less able to perform solid comparisons and more prone to follow established practice rather than risking the application of new, yet unproved, technology even if the new technology could yield substantial benefits when introduced widely. Only R&D personnel trained at the University on a number of alternatives, including radical new ones, can be the best vectors of evolution and innovation.

3.2 Limitations

Software Emphasis. Even if the Artist consortium evolved recently to include more hardware experts, our proposal addresses mainly the software aspect of the embedded system domain as this is the root of the Artist consortium.

Undergraduate Degrees. A comprehensive approach to education in embedded systems should include undergraduate courses. However, it is more difficult to assess the needs of undergraduate students in such a diverse cultural milieu than the ones of graduate students. Hence we decided to limit ourselves to graduate curricula. We note that two different approaches to undergraduate education are possible: one is to identify a subset of the topics identified for the graduate curricula, the other is to include all subjects but at a shallower level. By the same token, both deductive and inductive approaches could be followed for this kind of education. Dedicated experiments should be designed to define the best approach. At this time, we do not feel comfortable recommending a particular approach for undergraduate curricula.

Level of Abstraction. Due to education diversity, a curriculum rigidly organized into courses and modules would not be appropriate. For this reason, we propose a curriculum defined in terms of bodies of knowledge: what is important is the knowledge students will finally acquire, not how and when this knowledge is acquired. These bodies of knowledge are larger than individual courses. This makes agreement easier and allows a larger degree of freedom for practical implementation. Furthermore, according to the emphasis placed on different bodies of knowledge, one can get a quite large variety of curricula.

At this stage of generality, we do not distinguish between practice and theory as the emphasis on one versus the other gives an additional degree of freedom we leave to educators in defining a particular curriculum.

4. CURRICULUM DESCRIPTION

The scope of this paper is identifying large bodies of knowledge, which, we believe, are both useful and provide the foundations of the domain. Coherently organizing the bodies of knowledge identified is not an easy task, any organization will probably appear arbitrary, and will have unavoidable overlaps between different components. The curriculum organization is driven more by fundamental issues than by the structure of the application domains.

For each body of knowledge, we describe its pertinence in the curriculum, the required background, and its contents. The background is needed because we do not aim to describe a full curriculum, but only the components that concern embedded systems. We thus describe here a Master-level curriculum and we assume it is built upon a general-purpose computer science and engineering undergraduate background that we summarize in 4.1 and then mention when needed as background in other sections.

We chose not to overly detail the curriculum to allow flexibility. At the same time, we decided to provide enough details to make it useful and understandable for practitioners. We paid particular attention to cross-correlation across different bodies of knowledge with emphasis on dependencies that suggest a particular course sequencing.

We have identified five major bodies of knowledge that, we believe, constitute the main pillars of embedded system science and engineering. These are: basic control and signal processing, theory of computing, real-time computing, distributed computing, and evaluation and optimization of extrafunctional properties.

To these major bodies of knowledge we added two transversal themes: system architecture and design and applications.

4.1 Foundations of Computer Science and Engineering

We first provide a typical list of computer science and computer engineering skills that constitute a general-purpose background that we believe any graduate student interested in embedded system should have mastered in undergraduate studies.

- Basic algorithms, working knowledge of an imperative high-level programming language (e.g., Java, C^{++}, fluent use of C).
- Basic notions on logic gates, combinational and sequential circuits, simple processor architecture, input/output devices, interrupts, buses. Basic knowledge of assembly language.
- Elements of language theory (automata, context-free grammars, regular expressions).

- Implementation of programming languages (interpretation, compilation, mixed solutions). Lexical and syntactic analysis, static analysis, code generation, and optimizations.
- Basic operating systems, concurrent, and distributed programming.
- Software modeling, analysis and design, e.g., life cycle, testing methodologies, and tools.

Social impact and ethics of computer science and computer engineering may also be considered as part of the general background, even though this should not be specific to embedded systems but part of any solid engineering and computer science curriculum.

4.2 Basic Control and Signal Processing

Motivations. This is a relatively new item in a computer science curriculum; its presence is due to the fact that many applications deal with the control of physical processes and the processing of physical signals. The purpose here is not to train specialists in these topics, but to give computer engineers a minimal background in these aspects. In most of control applications, software runs in tight interaction with the physical environment and this body of knowledge is intended to sensitize students to the interactions among the physical and the computing world. This interaction points to global properties of utmost importance for a correct behavior of systems, such as stability and resonance. If the control system developer is not aware of the problems raised by this tight interaction, he/she may underestimate the importance of some physical details that may have serious consequences for the overall correctness of the design. In this area, we believe that training in the use of tools such as Matlab/Simulink that are becoming the *de facto* standards for specifying and simulating software controlled systems, is essential in providing the necessary virtual experimental set-up to appreciate the interaction between computing and control. Finally, many optimization techniques for resource allocation (energy consumption, for instance) are based on the principles of Control Theory concepts, such as feedback.

Background. A general scientific education based on solid foundations of physics and mathematics is required, with special emphasis on differential and integral calculus. Most computer scientists have a weak background in nondiscrete mathematics as computer science curricula emphasize abstractions that eliminate the physical world from consideration. We believe that to correctly address the issues arising from the design of embedded systems, students enrolled in computer science programs need a general continuous mathematics course to master the interaction between the physical world and the computer-based controllers typical of embedded systems.

Content. Modeling is a central activity and the most demanding one for control engineers. It is important that all embedded systems engineers are given an understanding of modeling from physical principles and modeling from data (learning models). Training should provide some notion of physical systems and signal modeling, such as linear time-invariant differential equations. Notions of Fourier and Laplace transforms are important to illustrate notions such as stability, bandwidth, and resonance and, in general, modeling uncertainties should be surveyed.

The notions of state and feedback are essential in control, and we believe they are also important for all aspects of embedded systems engineering; notions of feedback and continuous control should be provided, for instance, by teaching the basics of pole placement. Basic techniques for linear control design, including dealing with robustness and noise, should be proposed. More advanced difficulties, such as dealing with nonlinearities and delays, should be briefly discussed.

Sampling and sampled control should be addressed. This is important, as it builds a strong basis for computer implementations, such as time-triggered ones. Thus, Shannon sampling theory should be taught, and then sampled-data control systems should be introduced, along with z-transforms. In addition, the rationales for the choice of sampling periods should also be taught (this is seldom the case). Similarly, notions of (linear) digital filtering and fast Fourier transforms should be provided.

Important effects should be included that need to be considered when implementing control systems, such as quantization, different approaches for discretization, and time delays.

Discrete-event control is also important. It builds on almost the same material as language and automata theory of computer science and thus provides a natural bridge between the two disciplines. Discrete-event control favors computer implementations that are event-triggered systems. This is a problem when sampled (time-triggered) and event-triggered systems are to be mixed. The lack of sampling theory for event-triggered systems should be noted and the methods used in practice to compensate this deficiency should be explained.

Finally, some aspects of the emerging hybrid-control theory could be presented.

Documents and Teaching Material. Many textbooks and papers have been written on these topics. Of particular interest is:

- Edward A. Lee and Pravin Varaiya, 2002. *Structure and Interpretation of Signals and Systems*, Addison Wesley, Reading, MA (www.aw.com/info/lee/, ptolemy.eecs.berkeley.edu/eecs20/)

since it has been written in the context of what we argued in this paper. The course taught at Berkeley based on this book is briefly described in another paper in this issue. An important book also on this subject, but maybe more appropriate for teachers than for students given its level of sophistication, is:

- Karl J. Åström and Björn Wittenmark, 1996. *Computer-Controlled Systems—Theory and Design*, 1996, Prentice Hall, New York (www.control.lth.se/publications/books/astwit90.html)

Practice. Virtual experimentation of the techniques and of the behavior of control systems should be taught using popular tools such as Matlab/

Simulink/Stateflow or some of their open-source counterparts, such as Scilab/ Scicos (www.scilab.org) or Octave (www.octave.org).

4.3 Theory of Computing

Motivations. In a mirror situation with the previous body of knowledge, theory of computing is often neglected in engineering curricula specific to application domains. We believe that a well-balanced view of computing is necessary for any engineering curriculum that deals with embedded systems. The theory aspects are important since they provide a solid understanding of the behavior of computers and introduce techniques that can radically improve the safety and correctness of embedded systems, such as formal validation, verification, and testing.

Background. Basic knowledge of programming and algorithms are necessary to introduce the aspects of the theory of computing useful for the design of embedded systems.

Content. We believe that semantics is the foundation for a rigorous view of computing and we emphasize its role in the curriculum. There are several approaches to semantics of computing languages. We believe that the three semantics listed below are all necessary to build a strong background for embedded software: (1) Denotational semantics that builds upon functions and can be used to establish close links with basic control and signal processing.[1] (2) Axiomatic semantics that allows reasoning about programs. (3) Structural operational semantics that favors a point of view close to automata and machines.

The challenge is to present semantics in a unified framework. This part of the curriculum is instrumental to present important tools such as logics, induction, and fix-point theory.

The most relevant part of this body of knowledge is semantics of concurrent systems associated with temporal logics, verification, and synthesis techniques. In parallel, important theoretical aspects of computer science, such as computability and complexity, are useful to show students the limits of computing.

Finally, we should introduce fairly recent topics in the theory of computing that have been useful in the world of embedded computing such as: timed automata and their associated decision (analysis, synthesis) methods, and elements of cryptography and cryptographic protocols that are becoming important in many embedded systems (e.g., smart cards). As in the other parts of the curriculum, tools are an important component in our view of embedded system education. We believe that, in the teaching material about the theory of computing, assertion and property languages used in industrial tools, such as Sugar, should be included.

Finding common aspects and approaches with the control part of the curriculum should be emphasized.

[1]The standard formalisms for control engineering, such as, Matlab/Simulink, could be discussed here.

Documents, Teaching Material, and Practice. There is no textbook that includes all the topics listed above. Hence, the teaching material has to be extracted from a list of books, for example:

On semantics:

- Hanne Riis Nielson, Flemming Nielson, 1992. *Semantics with Applications: A Formal Introduction,* Wiley, New York (www.imm.dtu.dk/~riis).
- Glynn Winskel, 1993. *The Formal Semantics of Programming Languages: an introduction,* MIT Press, Cambridge, MA.

On computation:

- John E. Hopcroft and Jeffrey D. Ullman, 2001. *Introduction to Automata Theory, Languages, and Computation,* Addison Wesley, Reading, MA.

On concurrency:

- R. Milner, 1989. *Communication and Concurrency,* Prentice Hall, New York.

On verification:

- Z. Manna and A. Pnueli, 1995. *Temporal Verification of Reactive Systems: Safety,* Springer-Verlag.
- B. Bérard, M. Bidoit, A. Finkel, F. Laroussinie, A. Petit, L. Petrucci, and Ph. Schnoebelen, 2001. *Systems and Software Verification. Model-Checking Techniques and Tools,* Springer.

Practice. Hands-on experience with available modeling and verification tools, such as model checkers, static analyzers, and theorem provers, should be the backbone of the experimental part of the curriculum on the theory of computing.

4.4 Real-Time Computing

Motivations. Real-time computing includes all aspects of design and validation, languages, algorithms, programming, compiling, operating systems as applied to situations where real-time constraints are essential. This is clearly a core body of knowledge for embedded systems and is the most often taught in existing curricula.

Background. This broad subject requires some knowledge of classical computing in all its aspects.

Content. The curriculum should first introduce a taxonomy for real-time applications, for example, soft and hard real-time. Then, the following topics should be taught:

- *Operating systems*: We recommend to introduce here the fundamental issues of real-time operating systems, scheduling theory and schedulability analysis for real-time systems. It is useful to distinguish between time- and event-triggered systems and between preemptive and nonpreemptive scheduling. Basic scheduling algorithms should be taught, such as Table-Driven, Rate

Monotonic, Priority Inheritance and Priority Ceiling, and Earliest Deadline First.

- *Compilers*: Compilation techniques specific to real-time applications should be taught, either geared toward general-purpose computers or toward specific ones, such as DSPs. The problem of estimating worst-case execution time should also be addressed here.
- *Languages*: Dealing with time and concurrency in programming languages has led to the distinction between two paradigms:
  - —Asynchronous paradigm. This is based on "tasking" (ADA) or "threading" (Java), in which the control on task executions is provided by means of timers and priorities. Languages supporting this paradigm are associated with their own run-time model.
  - —Synchronous paradigm. This is based on logical time where the coordination of parallel activities is handled within the language itself. In principle, implementations may run on bare machines—without using any real-time operating system at all. Nevertheless, real-time operating systems may support synchronous execution. The design tools used in control theory, such as Simulink/Stateflow, are instances of this paradigm.

  We recommend in this part of the curriculum to underline the links with Computing Theory.
- *Design and validation*: Students should be trained to design real-time systems, from specification to partitioning the application into concurrent real-time tasks.

  The courses should adopt an appropriate level of abstraction. Although there is a need for standardization, a commonly accepted design methodology does not, as yet, exist. Here we recommend to show one or more methodologies depending upon course focus. For instance, some methodologies are more useful for given application domains, such as aerospace design, while others are more useful for codesigning embedded systems.

  Real-time component-based design should be introduced here, for example, introducing timing aspects in UML.

Documents and Teaching Material. Once more there is no single textbook that covers all the material of interest. We recommend to extract course material from the following books:

On synchronous languages:

- N. Halbwachs, 1993. *Synchronous programming of reactive systems,* Kluwer, Academic Publ., Boston, MA. (www.wkap.nl/prod/b/0-7923-9311-2)
- Stephen A. Edwards, 2000. *Languages for Digital Embedded Systems*, Kluwer, Academic Publ., Boston, MA. (www.wkap.nl/prod/b/0-7923-7925-X)

On asynchronous languages, real-time operating systems and scheduling:

- Alan Burns and Andy Wellings, 2001. *Real-Time Systems and Programming Languages (3rd Edition), Ada 95, Real-Time Java and Real-Time POSIX*, Addison Wesley.

On real-time operating systems and scheduling:

- Jane W. S. Liu, 2000. *Real-Time Systems*, Prentice-Hall, New York.
- Giorgio Buttazzo, 2005. *Hard Real-Time Computing Systems: Predictable Scheduling Algorithms and Applications*, 2nd Edition, Springer. (www.wkap.nl/prod/b/0-7923-9994-3)

In many cases, articles from real-time systems conferences and industry trade shows can provide good reading material on how to build real-time systems.

Practice. A basic outcome of the experimental part of the curriculum for real-time computing should be to show students that real-time systems introduce additional difficulties in system design and programming. It should include hands-on experience and comparisons between different languages and real-time operating systems, for instance, showing the pros and cons of different approaches on the same benchmark examples.

Real hardware platforms should be used to demonstrate how a failure to compute in real time can have visible side-effects. Good kits for laboratories for this task are the LEGO Mindstorms Robotics kits, which provide inexpensive and reliable hardware that can be afforded by most educational institutes. These systems were used with excellent results in a course on real-time software at Berkeley.

For teaching real-time operating-systems concepts, a simulated hardware environment can be used. For this purpose, the simulators provided with most embedded development environments or stand-alone products such as Virtutech Simics can be used. Students can then run actual real-time operating systems and experiment with scheduling policies, priorities etc. Research RTOS that run on PCs and workstations are:

- Shark (shark.sssup.it), developed at Scuola Superiore Sant'Anna that has been used as educational software for experimenting with scheduling algorithms in many courses in Italy and Sweden.
- MARTE OS (marte.unican.es), developed at University of Cantabria, that is also very well suited for teaching RTOS concepts, and that supports the Ada language natively.
- Open-source real-time OSes, such as RT-Linux (www. realtimelinuxfoundation.org) and its variants, or eCos (sources.redhat.com/ecos/).

4.5 Distributed Computing

Motivations. Distributed computing is another core competence for embedded system designers and, as such, should have an important place in any curriculum. All basic computing concepts such as languages, compilers, algorithms, and operating systems have counterparts in the distributed domain. In addition to these basic material, a strong link should be established with fault-tolerance, for two reasons: (1) all fault-tolerant implementations require some kind of redundancy, that is itself distributed, and (2) distribution exposes problems of imperfect communication and synchronization. To cope with these problems, some kind of fault tolerance is needed.

An important body of theory on "Distributed Fault-tolerant Systems" has been developed over the past 20 years. These results deserve better recognition and should be included in the curricula.

Background. This is a broad body of knowledge and requires some awareness of classical computing in all aspects mentioned above, combined with real-time computing.

Content

- *Distributed algorithms*: The design of distributed algorithms and systems should take into account the following: Computing and communication properties. Typical questions to answer are: Are the communication delays known? Are they bounded? Are clocks available? How precise are they? Faults which can impair computations and communications. Questions to ask are: Which faults have to be covered? What is their frequency?

  Various models for computing, communication, and faults should be presented. Note that the terminology in this domain differs slightly from the one used in Computing Theory and in Real-Time Computing. For instance, the terms "asynchronous", and "timed" have been used with different meanings; these differences should be clarified as to avoid confusion and misunderstandings.

  Various distributed protocols, e.g., point-to-point, network, and broadcast protocols, should be presented with their properties and limitations. Associated problems, e.g., atomicity, commitment, elections, consensus, logical/physical clock synchronization, and membership, should also be presented together with the most important approaches for their solutions. In particular the different models of time, such as dense real-time, sparse real-time, logical time, and vector time should be discussed. Since physical time-based models are built on clock synchronization (including external time sources such as GPS), deep understanding of the underlying theory is vital.
- *Networks*: This part of the curriculum should introduce and thoroughly describe the variety of networks and associated protocols that can be found in embedded systems, ranging from Internet to field buses and network-on-a-chip. Fundamental issues of today's state-of-the art real-time communication protocols, such as flow control, support for composability, flexibility and protocol latency should be discussed. The influence of distribution on scheduling, e.g., how to jointly schedule processor tasks and network communications should be discussed. Furthermore, security aspects need to be addressed, especially since wireless communication is becoming increasingly popular. Finally, smart transducer networks should be presented.
- *Design*: Real-time and space distribution introduce another dimension of complexity that cannot be mastered without a strong design discipline and architectural principles as everyone that has fought battles in the real world of embedded systems has learned at his/her expenses. Since the introduction of structure and hierarchical relationships represents a most promising

approach for understanding complex systems, emphasis should be placed on composability and interface design to allow an almost independent design of subsystems. Furthermore, implementation technologies such as VHDL and FPGA design should be discussed, because of their potential impact on the future of embedded systems. In addition, human factors and man machine interfaces should be discussed. Basic dependability concepts need to be introduced as well to discuss topics like error detection, Byzantine agreement, replica determinism, and TMR systems. Emphasis should be placed on determinism, in particular, in case of simultaneity. Since any fault-tolerant system must be based on a precisely defined fault hypothesis, basic knowledge about the fault hypothesis is important. Furthermore, certification aspects (e.g., DO-178B), design for safety, reliability modeling, and diagnosis should be introduced. Principles for distributed fault-tolerant architectures, successfully used in various application areas, should be discussed and classified according to their real-time and fault-tolerance features. For instance, time-triggered architectures should be included, as well as globally asynchronous locally synchronous (GALS) architectures. These are the most widely used for fault-tolerant distributed critical control systems.

Other popular architectures for industrial applications, such as those based on CAN and Ethernet networks, should be included. Finally, examples of soft real-time, high-availability systems such as telecommunication switches (Ericsson Erlang OTP and the Linux/TelORB system) should be presented.

- *Validation*: As mentioned above, distributed systems introduce another dimension of complexity in design and therefore in validation. For safety critical systems, experience shows that high dependability cannot be achieved by trial-and-error techniques, because the design process may not converge. This is why validation methods for distributed systems are important. Important links to computing theory (semantics, model-checking) can be drawn here.

Documents and Teaching Material. Books on the topic include:

- Nancy Lynch, 1996. *Distributed Algorithms*, Morgan Kaufmann Publishers. (theory.lcs.mit.edu/tds/distalgs.html)
- Paulo Verissimo and Luis Rodrigues, 2001. *Distributed Systems for System Architects*, Kluwer Academic Publ., Boston, MA. (www.navigators.di.fc.ul.pt/dssa/index.html)
- Hermann Kopetz, 2001. *Real-Time Systems Design Principles for Distributed Embedded Applications*, Kluwer Academic Publ., Boston, MA. (www.wkap.nl/prod/b/0-7923-9894-7)
- Sape J. Mullender (ed.), 1993. *Distributed Systems,* Addison-Wesley, Reading, MA.
- Herbert A. Simon, 1996. *The Sciences of the Artificial, Chapter 8: "The Architecture of Complexity,"* MIT Press, Cambridge, MA. (mitpress.mit.edu)

Practice. Practice should be based not only on simulation but also include experimentation on real platforms.

Experimenting only with workstations linked through a local-area networks is usually not sufficient to expose students to the intricacies of embedded systems. Real embedded platforms should be used in the laboratories. In case these platforms are not available, simulated networks and computers can be used to provide at least some sense of reality.

Another important goal of practice in this domain is to show students that distributed computing is several orders of magnitude more difficult than ordinary programming and this is why experimentation is important.

4.6 Evaluation and Optimization of Extrafunctional Properties

Motivations. With extrafunctional (a.k.a. nonfunctional) properties we mean all the properties that are not impacting *what* the system does but *how* it does it. Consumed power and energy, execution time, throughput, quality of service, and dependability are examples of extrafunctional properties.

Beyond the superficial differences between these properties, they share common techniques for assessment and optimization. These techniques should be taught to engineers and computer scientists regardless of the areas their future careers will bring them to: optimization and evaluation are base pillars of all engineering activity.

Background. This is an advanced theme as it is central to linking the abstract to the physical world: the techniques that have to be used to ensure a guaranteed level of quality are likely to depend heavily on the other themes introduced in the curriculum, such as "distributed systems," "real-time computing," and may be taught in these themes. It is also necessary to provide a background in optimization (operation research), probability, statistics, and queueing theory. This background can be part of the theme or a precondition.

Content. The subject includes several important subtopics:

- *Performance*: The student must be introduced to the basic models for performance evaluation, usually based on queueing theory. Then, soft real-time systems should be introduced as a generalization of hard real-time systems when uncertainty in the temporal characteristics of the system is present.

  An introduction to dynamic real-time systems must be provided. The theme should contain an overview of basic techniques for scheduling soft real-time applications in a dynamic environment. The concept of resource reservation should be introduced, both in the network (presenting algorithms such as WFQ, WF2Q+, SFQ), and in operating systems (Resource Kernel, CBS, Fair Scheduling).

  It is then possible to show the connection between the performance models and the scheduling algorithms, by analyzing the performance of example applications (for instance a multimedia streaming program) using the techniques mentioned above.

  Finally, the topic of QoS management and negotiation should be explained. In particular, it is important to introduce the concept of adaptive applications and adaptive systems. Basic on-line and off-line optimization techniques can be used to maximize the overall quality of the system. A survey of basic techniques such as imprecise computation, elastic scheduling, value-based scheduling, and adaptive reservation should be given together with an introduction to feedback scheduling techniques. In this last topic, there are interesting connections with control system theory.

- *Dependability*: This is a generic concept that encompasses several figures of merit, such as reliability, availability, safety, security, and maintainability. However, the techniques for assessing and improving each of these aspects are very similar and will not be distinguished at this level of detail. The more important techniques should be introduced: first combinational techniques (fault trees, reliability networks) and then dynamic ones based on Markov chains and derived techniques (stochastic Petri Nets, for instance). This approach allows the introduction of important aspects of fault tolerance, such as selective and massive redundancy, detection, replacement, and masking. It is important to note here the analogy with the techniques used for performance evaluation. Furthermore, links with Computing Theory and Control can be stressed by viewing Markov chains as stochastic automata and linear time-invariant differential systems. For instance, the analogy between state lumping in Markov chains and automata minimization is interesting, as well as the use of Laplace transforms for solving Markov chains.
- *Power consumption*: There is a growing body of techniques being proposed to reduce the power consumption of computer systems, not only in hardware, but also addressing software issues: power management and evaluation, techniques such as algorithmic tweaks to lower computational needs, parallelization to save power, turning off unused parts of a system, operating system techniques, such as measuring behavior, and architectures for low-power systems, including both hardware and software.
- *Memory and stack usage*: Due to the hard limits on the resources available in an embedded system, engineers must learn how to track and determine the amount of memory used by and needed by a system. Of particular interest is the size of stacks for software, since process stacks can occupy a significant amount of the available RAM in an embedded product. Techniques that need to be taught include static estimation of memory usage and dynamic techniques, such as high-water marking and profilers to determine memory usage. The introduction of stack measurement techniques that are present in many real-time operating systems is a good idea.
- *Execution time*: Determining the execution time of processes, tasks, or other units of computation is of utmost importance to the construction of embedded real-time systems. Students should be taught techniques applicable to both soft and hard real-time systems, such as using hardware clocks to time executions, watching bus transactions with logic analyzers or oscilloscopes, or using static analysis techniques. The support in current programming tools for time measurement (RTOS tool suites and stand-alone tools) should be included in the curriculum.
- *Trade-offs*: In general several of the points listed above are conflicting, for instance, power consumption and performance. Hence, methods for managing trade-offs in multivariable optimization should be addressed here.

Documents and Teaching Material. Some books included:

- R. Ramakumar, 1993, *Reliability Engineering: Fundamentals and Applications,* Prentice Hall, New York.
- Giorgio Buttazzo, Luca Abeni, Marco Caccamo, and Giuseppe Lipari, 2005. *Soft Real-Time Systems: Predictability vs. Efficiency*, Springer.
- Jean-Claude Laprie, 1992. *Dependability: Basic Concepts and Terminology*, Springer Verlag.
- Robin A. Sahner, Kishor S. Trivedi, and Antonio Puliafito, 1996. *Performance and Reliability Analysis of Computer Systems: An Example-Based Approach Using the SHARPE Software Package (The Red Book)*, Kluwer Academic Publ., Boston, MA. (www.ee.duke.edu/kst/book2.html)

Practice. The Shark kernel (shark.sssup.it) provides several soft algorithms and libraries that can be used by the students to develop soft real-time systems, such as multimedia applications.

For the topic of performance evaluation, the student should play with reservation-based systems. It is possible to use Linux/RK (www-2. cs.cmu.edu/ rajkumar/linux-rk.html), a modification of the Linux kernel that implements a resource reservation scheduler. The student should be able to run soft real-time and legacy applications on this system. Possible lab projects are: implementation of a QoS-aware MPEG player and implementation of a QoS manager system that dynamically accepts new applications with a certain guaranteed quality of service

For Timing-Analysis, students should be exposed to the tool aiT of AbsInt (www.absint.de/ait/) that provides one of the most accurate software estimation approach known today.

For evaluation of extrafunctional quantities, the students could use Metropolis (www.gigascale.org/metropolis) that, with its quantity managers, can handle any of them as long as an appropriate algebra to manipulate them is provided.

To understand dependability, students should be exposed to the difficult problem of modeling and computing very small probabilities, as required in very critical systems, and the need for a very careful design method.

4.7 Putting It Together: System Architecture and Engineering

Motivations. Embedded system design requires connecting in a unified framework all the bodies of knowledge presented in the previous subsections. The integration effort is certainly not a trivial task and merits to be treated as a separate body of knowledge on its own. The essential aspect of this important topic is the combination of heterogeneous objects and objectives. For instance, building a system that is distributed, fault-tolerant, and real-time, is a problem in itself, and the solution is a multifaceted development method.

Furthermore, these development methods need to cover the entire life cycle of systems: project management, problem understanding and requirements specification, analysis and design, implementation and validation methods, modification and maintenance.

In the curriculum, this topic must have a relevant position. It is probably best to leave it toward the end as a final reflection on the intricacies of embedded system design and on the need of rigorous methods for dealing with them.

Background. All the bodies of knowledge presented in the previous subsections.

Contents. This theme should stress the need for system architecture, considered as a way of fulfilling the multifaceted requirements of an application by coherent organization. The notion of platforms as backbones for architectural design should also be introduced.

Design methodologies that encompass the entire design process from conception to final implementation are the cultural underpinnings of this part of the curriculum. Several methodologies have been proposed over the years: some cover the entire design process, see, for example, POLIS (www-cad.eecs.berkeley.edu/Respep/Research/hsc/abstract.html, COSYMA (www.ida.ing.tu-bs.de/research/projects/cosyma), and some parts of Metropolis (www.gigascale.org/metropolis).

Faithful to our principles, we give here a slant toward the software angle of system level design and mention promising methodologies for embedded software. Architectural styles, UML for RT, and similar, are examples of techniques used for software design. Top-down approaches and structural programming, object-oriented and component-based approaches are all mainstays of good programming techniques that have an important role in system design. The implementation part should include principles for writing good programs (general, efficient, modifiable, modular). Verification is a very important aspect of system design: functional testing and requirement validation should be taught. Finally, maintenance procedures, software, and process measurements should be included.

Basic concepts of component-based development should be included: Component specification, software architecture and component-based approaches, designing component-based system, designing components as reusable entities, component specification, component evaluation and certification, and component testing. In addition, the component-based development process such as software product lines should be included. More advanced topics are component compositions, system emerging properties, and component properties. The RT and embedded specific part includes the following topics: properties and interfaces of real-time components, composition and predictability of RT, and other extrafunctional properties of component-based systems. An insight into different component technologies and middle-ware should also be included as a practical part: CORBA, RT CORBA, RT Java, and some of component models implemented in different domains of embedded systems.

Recently model-based design has caught the attention of the industrial community as a powerful design paradigm. In this approach, requirements are captured at a high level of abstraction with mathematical models. Implementation is seen as a refinement process that maps the models into implementation platforms possibly in an automatic way. The Mathworks with Real Time Workshop, Etas with ASCET, and dSpace with Target Link, offer automatic code generation from mathematical models of the desired behavior captured with Simulink or with ASCET models of computation. This approach is based on the “informal” nature of the models of computation in these domains, while code generation from models is the result of a rigorous method for synchronous models as the ones embedded in synchronous languages, such as Lustre and Esterel (www.esterel-technologies.com). Other approaches focus on heterogeneous models of computation, their capture, their refinement, and analysis. Environments such as Metropolis (www.gigascale.org/metropolis/) and Ptolemy (ptolemy.eecs.berkeley.edu/) may be introduced to show students how to capture heterogeneity and how to analyze it in a rigorous framework where the semantics of the models used are unambiguous.

The verification method that is used mostly today is simulation. Simulation is not only a technique used for early validation, but can be considered as a problem in itself in the embedded software domain. Indeed, software cannot be developed without some model of the environment, be it a physical process or a piece of hardware which is not already developed. Both cases require reliable and semantically well-defined simulation techniques. The environments mentioned above can serve the purpose of showing students the effect of combining heterogeneous models.

Another important aspect is related to the management of the development process. Different types of projects and different phases of a project should be introduced. Team work, specific roles in the project, supporting methods and tools should also be emphasized, possibly in courses where projects take an important part of the final evaluation of the student.

A nonexhaustive list of projects and examples include design of a distributed safety-critical control system; soft real-time QoS-oriented distributed application; and component-based product.

Documents and Teaching Material. Once more there is no all encompassing textbook that can be used to teach this part of the curriculum. Two books of interest are:

- Ivica Crnkovic and Magnus Larsson, 2002. *Building Reliable Component-Based Software Systems,* Artech House Publishers. (www.idt.mdh.se/∼icc/)
- Jim Cooling, 2003. *Software engineering for Real-Time Systems,* Addison Wesley, Reading, MA.

However, we recommend to access the recent literature on design methodologies and tools as published in a number of journals and conference proceedings. A book of recent publication that includes a number of important papers about embedded system design should also be consulted:

- Richard Zurawski (ed.) 2005. *The Embedded System Handbook*, CRC Press, Boca Raton, FL.

4.8 Applications

Practice is an essential component for a well-rounded education in embedded software and systems. A curriculum on embedded software and systems cannot rely only on theoretical courses. Laboratory work should be used to illustrate each course.

Beyond illustrating each course individually, it is important to tie them together through common, multicourse projects possibly in synchrony with the course on system design and architecture. In a curriculum such as the one described here, students are taught many different theories and methods. Practice is important to demonstrate their joint applicability for real systems. This will provide students an understanding of their practical interest. Practice, especially in the form of larger projects, gives students the opportunity to integrate the different pieces of acquired knowledge.

Practical laboratories also offer students a chance to become familiar with embedded systems programming tools, such as cross-compilers, various debugging tools such as emulators, simulators, JTAG-probes, bus analyzers, and logic analyzers, together with more modern design tools such as Simulink/Stateflow, RTW, ASCET, Esterel, Lustre, Ptolemy II, Metropolis, and Co-Ware. It would be also important to use reconfigurable platforms such as Xilinx Vertex II Pro that features a multiprocessor and FPGA architecture (four Power cores) together with its development tools that are ideal vehicles to deploy a design on a computationally rich implementation platform.

This helps the student understand problems specific to resource-constrained environments, the importance of choosing the right platform and the ways in which the system architecture affects the runtime characteristics.

To implement this section, a wealth of projects should be available that, on one hand, are complex enough to expose the several aspects of the domain and the links between them and, on the other, simple enough to allow the student to complete the design and to demonstrate the results of their work. We recommend to choose realistic projects that involve application domains ranging from multimedia to transportation systems, from white goods to monitoring, and control systems.

5. CONCLUSION

We, the Artist Education working group, presented guidelines for the development of a graduate curriculum for embedded software and systems. Two aspects of our approach should be emphasized to place our work in context:

- The diversity of educational systems and approaches complicates the elaboration of concrete and detailed curricula. For this reason, only guidelines and abstract curricula have been proposed.
- The Artist consortium was initially focused on embedded software; for this reason, the curriculum has only a few remarks about competencies in hardware and electrical engineering.

We nonetheless proposed contents that, we believe, should be found in a curriculum in embedded software and systems. Our choices were driven by fundamental issues, points that are both difficult and important to the subject and which are not likely to be easily acquired through on-the-job training. This has led us to identify five pillars of fundamental knowledge on which to build a curriculum:

- Control and signal processing, required for applications where the embedded device acts as a controller on the physical environment.
- Computing theory (algorithms, methods, and tools for formal description and analysis including validation of computing devices), complementary to control and signal processing - a body of knowledge that focuses on the interaction with the physical environment. The interaction of this and the previous topic is the cross-road for embedded system design foundation.
- Real-time, composed of several approaches that should be covered and compared with a critical and synthesis point of view.
- Distributed systems, also core knowledge in the field. This is not always satisfactorily covered, although it provides answers to relevant and difficult problems. This is due to the fact that the theory has been developed by the relatively separate "Distributed and Fault-Tolerant" communities.
- Optimization and evaluation, including traditional engineering methods and tools for measuring, evaluating, and optimizing extrafunctional properties specific to embedded systems: such as dependability, performance, power consumption, and weight.

In addition, two transversal themes were emphasized:

- System architecture and engineering providing rigorous design methods and tools for building systems meeting given requirements. These approaches should rely on the above bodies of knowledge.
- Applications including methodologies and tools that are the experimental foundations for the curriculum and where students can learn the power and limitations of the theory and methods presented.

To the best of our knowledge, no existing curriculum fully implements our proposal, at least in Europe. There are several reasons for this situation; we list some of them here:

- The concept of a curriculum is not well-defined or equally understood everywhere. It depends on the local education system and on the teaching institution. In many cases, students are free to choose the courses they want, provided they follow prerequisites and their program is accepted by their supervisor. Thus, identifying a precise curriculum is difficult.
- The educational system has significant inertia and takes time to adapt to evolving industrial and cultural necessities. In particular, it takes time to organize curricula and gather specialists from the required disciplines. This is a major point: we believe that only specialists can teach the subtleties of the proposed themes and have a sufficiently broad vision of the subject to draw useful links between the different subjects addressed here.

As a consequence, the lack of adequately trained engineers has an impact on current industrial practices. The lack of specialists fully aware of the various design techniques, induces fragmentation, inefficiencies, and difficulties in communication and shared experience within a company. For instance, aeronautics and space, that are closely related fields and address similar problems, use different methodologies, design flows, and tools, for reasons that do not seem at all obvious.

Another consequence is fragmentation of research. Some examples have been cited in this document. For instance, very basic terminology related to timing and communication, words such as "timed", "untimed," "synchronous" and "asynchronous," do not have the same meaning for the Language Theory communities as they have for the Distributed and Fault Tolerant Systems community. Here also, this lack of a common cultural ground results in communication difficulties and fragmentation of research. Moreover, this is a circular phenomenon: since teachers do not understand each other, neither do students. Papers are poorly understood, conferences are separated, and, finally, communities do not interact.

Education is clearly a place where these issues can be addressed. For instance, some differences in practices in different industrial segments can be justified for cultural reasons. Training students to experiment and compare several approaches can be a good way to bridge gaps. This is why we believe that the proposed abstract curriculum provides a basis for avoiding these drawbacks and is worth being considered.

Gaining acceptance in the academic community is essential for this proposal to help in overcoming the difficulties presented above. We are fully aware of the limitations of our proposal, yet we believe it constitutes a basis for dissemination, debate and refinement. Some recommended that the document serve as a basis for some sort of European "certification" that would assess compliance of curricula to this document. This is clearly an interesting possibility, albeit politically difficult to implement. Yet, just reaching agreement on its content, among colleagues and academic authorities is, in itself, a worthy goal.

Received September 2004; revised January 2005, March 2005; accepted May 2005