Cent. Eur. J. Comp. Sci. • 1(1) • 2011 • 108-116
DOI: 10.2478/s13537-011-0008-y

VERSITA

Central European Journal of **Computer Science**

A vision for computer science – the system perspective

Vision Article

Joseph Sifakis*

Verimag Laboratory, Centre Equation,
2 ave de Vignate, 38610 Gieres, France

Received 18 Feb 2011; accepted 03 Mar 2011

1. The evolution of computer science

Computer Science is a young discipline. Its foundations were laid in 1936 by the seminal work of A.M. Turing and K. Gödel. Its scope and focus have changed continuously over the past seven decades. The first computers ran numerical software for defense applications. In the 70's, the advent of mainframe computers broadened the scope to include commercial applications. In parallel, large-scale circuit integration allowed exponential increases in computing power (Moore's law). In the 80's, the convergence between information technologies and telecommunications opened the way for the Internet, the Web and the Information Society. In the 90's, another very important, but less visible, revolution started with the dissemination of embedded systems technologies. More than 95% of the chips produced today are for embedded applications. These are electronic components integrating software and hardware jointly and specifically designed to provide given functionalities, which are often critical. They are hidden in devices, appliances and equipment of any kind: mobile phones, cameras, home appliances, cars, aircraft, trains, medical devices etc. In 2008, the average person used about 230 embedded chips every day: 80 chips in home appliances, 40 chips at work, 70 chips in cars, 40 chips in portable devices.

In the near future, another anticipated, important landmark will be the advent of the Internet of Things as the result of the convergence between embedded technologies and the Internet. The idea is to use internet technologies to integrate services provided by hundreds of billions of embedded systems. This will require an upgrade of the internet infrastructure to make it more secure, safer and reactive. Current features for exchanging multimedia documents will be extended to encompass real-time monitoring and control. Systems are becoming ubiquitous: the state of almost everything can be sensed, measured and monitored; people and objects can communicate and interact in entirely new ways; intelligent systems allow enhanced predictability of events and optimal use of resources.

It is hard to imagine what Computer Science will be in two decades. More than any other discipline, it is driven by applications and exponential progress in technology. The broadening of its perimeter is accompanied by a shift in focus from algorithms and programs to systems.

* *E-mail: Joseph.Sifakis@imag.fr*

2. From programs to systems

I see two main landmarks in the evolution of CS: the advent of Computers and the development of Theory of Computing. The first allowed the move from mechanical computational devices to much faster and reliable automated electronic circuits. The second opened the way for mechanizing computation by developing models of computation for studying algorithms, programs and their properties. The central issue addressed is whether, and when, functions can be computed by using models of computation. A remarkable fact about these models is that they ignore physical time and resources. Computation is a finite sequence of steps corresponding to the execution of primitive operations. Complexity theory is based on abstract notions of time and memory. Programs and algorithms are just relations independent from the physical resources needed for their execution. Their behavior is terminating, deterministic and platform-independent.
In contrast to programs and algorithms, systems are reactive; that is, they continuously interact with an external environment. Their inputs are stimuli that trigger state changes and computation of outputs that may modify the state of their environment. System behavior can be modelled as a relation between histories of inputs and histories of outputs. Systems are, in general, non-terminating and non-deterministic. Furthermore, their behavior is, in general, platform-dependent as their correctness depends on the dynamic characteristics of the execution platform e.g. execution times. Theory of computation is, by its nature, of little help for studying systems. Even if we perfectly understand the properties of a program and the properties of a hardware target platform, we have no theory to predict the behavior of the program running on the platform. The observed shift of focus in Computer Science from programs to systems should be accompanied by research for extending the Theory of Computing. Models of computation should be enriched to take into account physical resources and the interplay between systems and their physical environment.

3. Trends in system design

Embedded Systems break from traditional computing systems such as desktop computers and servers. They must jointly meet technical requirements such as:

- *Reactivity*: responding within a known and bounded delay. This is essential not only for real-time applications but also for efficient quality of service control.
- *Autonomy*: providing continuous service without human intervention. In particular, this means no manual restart but also optimal power management for portable devices.
- *Dependability*: invulnerability to threats including attacks, hardware failures, software execution errors.
- *Scalability*: performance increase is commensurable with the increase of resources.

In addition to these requirements, embedded systems must meet requirements for optimal cost/quality as they are integrated in mass-market products. Seeking an economic optimum in system design is much harder than achieving high quality without cost-effectiveness constraints.
Embedded technologies challenge our capacity to develop systems of guaranteed functionality and quality at acceptable costs.
Today we master, at high costs, two types of systems which are difficult to integrate: 1) safety and/or security critical systems of low complexity such as flight controllers or smart cards; 2) complex best-effort systems such as telecommunication systems and web-based applications. Dependability is the main concern for critical systems while best-effort systems seek optimal use of resources at acceptable levels of quality of service.
For future systems, we urgently need technology for

- *developing affordable critical systems*. In many application areas such as transport, health and energy, embedded technologies could be used to provide new services with significant impacts on quality of life and efficient resource management. For instance, drive-by-wire and brake-by-wire in cars could lead to lower production and operational costs by replacing “passive safety” with “active safety”.
- *safe integration of heterogeneous systems-of-systems*. The vision is to develop global services by integrating features provided by geographically distributed systems with different technical characteristics and using various

communication media. A main issue for strong integration is mastering interaction of critical and non-critical features, and error containment. How to prevent failures of non-critical services from affecting the behavior of critical services raises difficult problems which lack theory to be tackled with.

There exist several incarnations of the systems-of-systems vision. The most general one is the internet of things intended to develop global services by interconnecting everyday objects. One instance of this vision is smart grids for efficient and reliable energy management. Another instance is intelligent transport systems to improve safety and reduce vehicle wear, transportation times, and fuel consumption.

I believe that we cannot achieve any significant progress in these directions without considerable research effort into developing foundations for rigorous system design. Despite progress in theory, methods and tools over the past decades, there is an increasing gap between 1) the possibilities offered by progress in VLSI and telecommunications technology such as many-core chips and sensor networks; 2) the state-of-the art in system design and integration. Today, research in Computer Science lags far behind these needs. With a few exceptions [1, 2, 12, 13], the importance of system design is seldom recognized as a priority in research agendas and roadmaps.

4. Three grand challenges

Rigorous system design raises three Grand Challenges: 1) Marrying Physicality and Computation; 2) Component-based Design; 3) Adaptivity.

4.1. Marrying physicality and computation

We need theory and models encompassing continuous and discrete dynamics to predict the global behavior of a system interacting with its physical environment [1, 2].
The development of application software and its implementation must take into account constraints from:

- the *physical resources* of the hardware execution platform so as to optimize their use. This requires mastering interaction between software execution and the underlying hardware and, in particular, the impact of software design choices on the dynamic behavior of the circuit.
- the *physical environment* of the system. These are often user-defined, real-time properties such as deadlines and jitter.

We lack theory, methods and tools for tackling these problems. We need to revisit and revise computing to integrate paradigms from Electrical Engineering and Control Theory. To achieve this aim, we should be able to consistently combine models of computation with analytic models used in physical systems engineering. These describe the behavior of electromechanical systems with sets of differential equations. Engineers extensively use linear approximations represented as data-flow networks of interconnected components characterized by their transfer function. Components transform concurrent input data flows into output data flows. In contrast to these models, models of computation are procedural and inherently sequential. Interaction between components is control-flow e.g. by method call. We lack theory unifying the two types of models. Existing unifications such as hybrid systems theory [3] establish relations at semantic level (at transition system level). They cannot be used to relate analytic models and programming models through structure-preserving translations. For instance, system engineers need techniques for consistently combining formalisms such as Matlab/Simulink[1] and procedural programming languages.

4.2. Component-based design

We need theory, models and tools for the cost-effective building of complex systems by assembling heterogeneous components. This is essential for any engineering discipline. It confers numerous advantages such as productivity and correctness.

[1] *http://www.mathworks.com/products/simulink/*

System designers deal with heterogeneous components, with different characteristics, from a large variety of viewpoints, each highlighting different dimensions of a system. They often use several semantically unrelated formalisms e.g. for programming, hardware description and simulation. This breaks the continuity of the design flow and jeopardizes its coherency. System development is often decoupled from validation and evaluation.
System descriptions used along a design flow should be based on a single semantic model to maintain its overall coherency by guaranteeing that a description at step $n+1$ meets essential properties of a description at step n. The semantic model should be expressive enough to directly encompass component heterogeneity. Three different sources of heterogeneity exist [1]:

- *Heterogeneity of computation*: The semantic model should encompass both synchronous and asynchronous computation to allow, in particular, modeling mixed hardware/software systems.
- *Heterogeneity of interaction*: The semantic model should enable natural and direct description of various mechanisms used to coordinate execution of components including semaphores, rendezvous, broadcast, method call, etc.
- *Heterogeneity of abstraction*: The semantic model should support the description of a system at different abstraction levels from application software to its implementation.

Existing theoretical frameworks for composition are based on a single operator e.g., product of automata, function call. Poor expressiveness of these frameworks may lead to complicated designs: achieving a given coordination between components often requires additional components to manage their interaction [4]. For instance, if the composition is by strong synchronization (rendezvous), modeling broadcast requires components for choosing the maximal amongst several possible strong synchronizations. We need frameworks providing families of composition operators for the natural and direct description of coordination mechanisms such as protocols, schedulers and buses. These should encompass a unified composition paradigm for describing and analyzing the coordination between components in terms of tangible, well-founded and organized concepts. In addition to that, they should be equipped with tractable methods for ensuring correctness-by-construction, to avoid the limitations of monolithic verification. These methods use two types of rules [5]:

- *Compositionality* rules for inferring global properties of composite components from the properties of constituent components e.g. the composition of deadlock-free components is, under some conditions, a deadlock-free component. Compositionality rules should, in particular, take into account the emergence of new properties. For instance, proving mutual exclusion for sharing a resource in a composite component under the assumption of atomicity of actions of the constituent components. A special and very useful case of compositionality is when a behavioral equivalence relation between components is a congruence [6]. In that case, substituting a component in a system model by a behaviorally equivalent component leads to an equivalent model. Today, we lack compositionality theory for progress properties as well as extra-functional properties.
- *Composability* rules ensuring that essential properties of components are preserved when they are used to build composite components. Consider, for instance, two components: one is the composition of a set of components sharing a common resource accessed in mutual exclusion; the other is obtained as the composition of the same set of components scheduled for optimal use of the shared resource. Is it possible to obtain a single component integrating this set of components and such that both mutual exclusion and the scheduling constraints hold? System engineers face this type of non-trivial problem every day. They use libraries of solutions to specific problems and they need methods for combining them without jeopardizing their essential properties. Feature interaction in telecommunication systems, interference among web services and interference in aspect programming are all manifestations of the lack of composability.

4.3. Adaptivity

Systems must provide a service meeting given requirements in interaction with uncertain environments. Uncertainty can be characterized as the difference between average and extreme system behavior. Non-determinism of physical environments increases uncertainty. Furthermore, execution platforms have varying execution times due to layering, caches, speculative execution and variability due to manufacturing errors or aging.

Uncertainty directly affects the predictability of analysis techniques. Predictability is the degree to which a correct prediction of a property can be made either qualitatively or quantitatively. Due to uncertainty, system models represent safe abstractions of the actual system behavior and may include additional unfeasible execution sequences. Furthermore, for a given system model, exact analysis techniques are impossible due to non-computability of all essential system properties. For example, due to uncertainty, execution times of an instruction may vary between a best-case-execution-time (BCET) and a worst-case-execution-time (WCET) depending on the location of data e.g. cache memory or main memory and their size [7]. Due to uncertainty, WCET may be a hundred times larger than BCET. WCET and BCET cannot be exactly computed. Timing analysis tools provide only upper bounds and lower bounds, respectively. The quality of these approximations may be very poor.
Uncertainty and lack of predictability have a deep impact on system design methods and increasing development costs. Currently, there exist two diverging system design paradigms.

1. *Critical engineering* is based on worst-case analysis of all the potentially dangerous situations. Designers reserve statically all the resources (memory, time) needed for safe operation, leading to over-dimensioned systems. The amount of physical resources may be some orders of magnitude higher than necessary. This incurs high production costs but also increased energy consumption.

   For example, response times of tasks in a real-time system must be less than a given deadline. The response time of a task is computed from safe approximations of WCET of its statements. As a result, hardware platforms for hard-real time systems are extremely over-dimensioned.

   Another principle from critical engineering consists in using massive redundancy to enhance reliability. Redundancy techniques (e.g. TMR), lead to over-dimensioning and can be advantageously replaced by smart lightweight monitoring and error recovery techniques.

2. *Best-effort engineering* is applied to complex non-critical systems. It is based on average case analysis and dynamic resource management. Designers apply QoS management techniques for optimizing speed, memory, bandwidth, and power. Physical resources are dimensioned for availability in nominal cases. In critical situations, e.g., spike of service demands, the system service may be degraded or denied.

The separation between critical and best-effort systems is a means for coping with non-predictability by focusing on essential properties for each class of systems. Nonetheless, most applications integrate both critical and best-effort components. This raises multiple technical difficulties as attested by numerous problems experienced by car manufacturers over the past decade.
Two avenues are anticipated to overcome current limitations of the state-of-the-art:

- One is to enhance predictability through determinism. The key idea is to reduce intrinsic and estimated uncertainty by simplifying hardware architectures or by enforcing a time-deterministic observable behavior [8] e.g. by using time-triggered architectures [9]. I do not believe that this is a realistic and viable approach. Predictability is ensured by seriously penalizing performance. Furthermore, it requires significant changes in the way application software is written [14].

- The other is to bridge the gap between critical and best-effort engineering by applying adaptive control techniques to systems integrating both critical and best-effort components sharing global resources which are predictably sufficient for satisfying critical properties. The main idea is to use an Adaptive Controller which manages the resources dynamically so that critical properties are satisfied by assigning them a higher priority. The remaining resources can be used to satisfy best-effort properties. This avoids costly static resource reservation and does not require a very fine analysis to determine worst-case situations.

An Adaptive Controller monitors the state of a system and steers its behavior to meet given requirements specified as a set of objectives including hard constraints such as deadlines as well as constraints expressing the optimal use of resources. It combines three hierarchically structured functions: Learning, Objective Management and Planning. Depending on the state of the controlled system, the Objective Manager chooses an objective that meets the requirements. The objective is passed to the Planning Function. The latter computes an execution plan used to drive system evolution. The Learning Function is used to compute good estimates of parameters involved in the constraints on the Objective Manager e.g. worst-case and average values of parameters such as execution times and throughput.

Adaptive control was initially studied in Control Theory [10]. It finds an increasing number of applications in computing systems e.g. for quality control in multimedia systems, throughput control in networks, and self-maintenance and recovery in distributed systems.
An important issue in adaptive systems design is reducing the overhead due to monitoring and control. We need execution platforms that give up control to the application software for efficient resource management.
Adaptivity encompasses a new and realistic vision for "intelligent systems" quite different from the old vision of Artificial Intelligence. The latter considered that human intelligence can be so precisely described that it can be matched by a machine. Adaptivity is a means to enforce correctness in the presence of uncertainty by using control-based techniques.

5. A vision for computer science

5.1. Computer science as a discipline

Computer Science is a scientific discipline in its own right with its own concepts and paradigms. It deals with problems related to the representation, transformation and transmission of Information. As such, it studies all aspects of computing from models of computation to the design of software and computing devices.
Information is an entity distinct from matter and energy. It is a resource that can be stored, transformed, transmitted and consumed. It is immaterial but needs media for its representation by using languages characterized by their syntax and semantics. It should not be confused with physical information measured as entropy in Information Theory and Physics.
Computer Science is not merely a branch of Mathematics. As any scientific discipline, it seeks validation of its theories on mathematical grounds. But mainly, and most importantly, it develops specific theory intended to explain and predict properties of computation which can be tested experimentally.

5.1.1. Computer science vs. physics

Embedded Systems bring Computer Science closer to Physics. Marrying physicality and computation requires a better understanding of differences and points of contact between them. Is it possible to define models of computation encompassing quantities such as physical time, physical memory and energy? Significant differences exist in the approaches and paradigms adopted by the two disciplines.
Physics is based on continuous mathematics while Computer Science is rooted in discrete non-invertible Mathematics. Physics studies a given "reality" and tries to discover laws governing physical phenomena while computing systems are human artifacts. Its laws are declarative by their nature.
Physical systems are specified by differential equations involving relations between physical quantities. The essence of many physical phenomena can be captured by simple linear laws. They are, to a large extent, deterministic and predictable. Synthesis is the dominant paradigm in physical systems engineering. We know how to build artifacts meeting given requirements e.g. bridges or circuits, by solving equations describing their behavior. In contrast, state equations of very simple computing systems, such as a RS flip-flop, do not admit linear representations in any finite field. Computing systems are described in executable formalisms such as programs and machines. Their behavior is intrinsically non-deterministic. Non-decidability of their essential properties implies poor predictability. Synthesis is an intractable problem. Computing systems engineering relies to a large extent on verification and testing.
Computer Science enriches our knowledge with theory and models enabling a deeper understanding of discrete dynamic systems. It proposes a constructive and operational view of the world which complements the classic declarative approach adopted by Physics.

5.1.2. Artificial vs. natural intelligence

Living organisms intimately combine interacting physical and computational phenomena that have a deep impact on their development and evolution. They share several characteristics with computing systems such as the use of memory, the distinction between hardware and software, the use of languages. However some essential differences exist. Computation in living organisms is robust, has built-in mechanisms for adaptivity and, most importantly, it allows the emergence of abstractions and concepts.
I believe that these differences delimit a gap that will never be filled by computing systems. Consider simply robustness which means that the effects of small changes in a system are commensurably small. Discreteness makes practically impossible this property for existing models of computation.

Despite these differences, there exist today many opportunities for interactions and cross-fertilization between Computer Science and Biology. For instance, results from neuromorphic and cognitive computing could inspire new non von Neumann paradigms. Conversely, synthetic biology could benefit from existing CAD and systems engineering technology.

5.2. Research in computer science

Unfortunately, the current scope and focus of research in Computer Science fail to address basic problems raised by system design and engineering. Three syndromes determine choices and priorities of research communities.

5.2.1. The business as usual syndrome

Following beaten tracks rather than taking the risk of exploring new ideas is a prevalent attitude of researchers in all scientific communities. Research in Computer Science is not exempt from this syndrome.

5.2.2. The hype syndrome

More than in other disciplines, research in Computer Science has been over-optimistic regarding the possibility to solve hard problems and overcome obstacles. This can probably be explained by strong demand and incentives for innovation by funding agencies as well as strong push from applications and market needs.
Very often scientific roadmaps and position papers present "challenges" that are mere visions and take desires for reality. A scientific challenge is a designated and significant obstacle to the development of knowledge and its effective use in a given area [11]. It is distinct from incremental scientific progress, in that there is a clear break between before and after the accomplishment. It implicitly has some "positive" connotation as it must comply with the established ethical and public interest criteria. Finally, it is different from a vision which also represents a long-term objective with a broader scope but fails to meet at least one of the following criteria:

- *well-defined* either as a single problem (e.g. Fermat's Last Theorem) or a set of strongly related problems for which a framework is needed to provide the basis for tackling them (e.g. relativity theory).
- *plausibile*, that is it takes into account the existing body of knowledge, including well-established theoretical limitations e.g. complexity, computability and does not contradict or ignore experimental evidence.
- *relevant*, that is the resources needed for its achievement are proportionate to the importance of its goals and the risks for feasibility. Nonetheless, challenges can (and should) be formulated without taking into account the resources necessary for their accomplishment.

All of the following were once hyped as main breakthroughs: Artificial Intelligence, Fifth Generation Computers, Program Synthesis, True Concurrency, Web Science. Each item of this list fails to meet at least one of the above criteria. Most are not well-defined and others are simply not plausible due to theoretical limitations.

5.2.3. The nice theory syndrome

The proper goal of theory in any field is to make models that accurately describe real systems. Models can be used to explain phenomena and predict system behavior. They should help system builders do their jobs better.
A very common attitude is to work on mathematically clean theoretical frameworks no matter how relevant they can be. Paul Krugman Nobel Prize in Economics prize says, *"... in the academic world the theories that are more likely to attract a devoted following are those that best allow a clever but not very original young man to demonstrate his cleverness."*
Very often in Computer Science simple mathematical frameworks attract the most brilliant researchers who produce sterile "low-level theory" that has no point of contact with real computing. This leads to a separation between theoretical and practical work harmful for the discipline. To quote Einstein, *"Make everything as simple as possible, but not simpler".*
The opposite attitude is also observed. Frameworks exist intended to describe real systems such as UML and AADL constructed in an ad hoc manner. These include a large number of semantically unrelated constructs and primitives. It is practically impossible to obtain rigorous formalizations and build any useful theory for such frameworks.
We need theoretical frameworks that are expressive enough to directly encompass a minimal set of high level concepts and primitives for describing systems and amenable to formalization and analysis. As Saint-Exupéry said, *"Perfection is reached not when there is no longer anything to add, but when there is no longer anything to take away".*

Is it possible to find a mathematically elegant and still practicable theoretical framework for computing systems? As explained, due to fundamental differences, we cannot expect to have theoretical settings as beautiful and powerful as for physical systems. There is also a more profound reason: computing systems are human artifacts while the physical systems are the result of a very long evolution. When you throw a stone it will describe a parabola. There is no such an intelligible law governing the execution of a program.
Einstein used to say, *"The most incomprehensible thing about the world is that it is at all comprehensible."*
Computer Science deals with building artifacts. The key issue is constructivity, that is, the ability to effectively build correct systems [1]. As system synthesis from requirements is intractable for complex systems, we should study principles for building correct systems from components. The aim is to avoid *a posteriori* monolithic verification as much as possible. There already exists a large body of constructivity results in Computer Science such as algorithms, architectures and protocols. Their application allows correctness for (almost) free. How can global properties of a composite system be effectively inferred from the properties of its constituents? This remains an old, open problem that urgently needs answers. Failure in bringing satisfactory solutions will be a limiting factor for system integration. It would also mean that Computer Science is definitely relegated to second-class status with respect to other disciplines.

5.3. Teaching computer science

Computer Science curricula seldom recognize the importance of systems and fail to provide a holistic view of the discipline. I have the following recommendations:

- Teach students how to think in terms of systems (design process, tools, interaction with users and physical environment). Computer Science curricula should be extended and enriched by including principles, paradigms, techniques from Control Theory and Electrical Engineering.
- Teach principles rather than facts (foundations, architectures, protocols, compilers, simulation, etc.). Very often, courses are descriptive and present details that can be acquired later as needed along professional life. Students should be prepared to deal with the constant change induced by technology and applications. They also should be kept aware of the limitations of existing theory of computing. Very often theory makes assumptions that oversimplify reality.
- Put emphasis on information and computation as universal concepts which are applicable not only to computers and provide the background for triggering critical thinking, understanding and mastering the digital world.

References

[1] Henzinger T.A., Sifakis J., The Discipline of Embedded Systems Design, COMPUTER, 2007, 40, 36-44
[2] Lee E.A., Cyber Physical Systems: Design Challenges, 11th IEEE International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC 2008) (5-7 May 2008, Orlando, Florida, USA) IEEE Computer Society, 2008, 363-369
[3] Alur R., Courcoubetis C., Halbwachs N., Henzinger T.A., Ho P.-H., Nicollin X., Olivero A., Sifakis J., Yovine S., The Algorithmic Analysis of Hybrid Systems, THEOR COMPUT SCI, 1995, 138, 3-34
[4] Sifakis J., A Framework for Component-based Construction, Aichernig B.K., Beckert B. (Eds.), 3rd IEEE International Conference on Software Engineering and Formal Methods (SEFM05) (7-9 September 2005, Koblenz, Germany) IEEE Computer Society, 2005, 293-300
[5] Bliudze S., Sifakis J., A Notion of Glue Expressiveness for Component-Based Systems, LECT NOTES COMP SCI, 2008, 5201, 508-522
[6] Milner R., A Calculus of Communication Systems, Springer-Verlag, Secaucus, NJ, USA, 1982
[7] Wilhelm R., Engblom J., Ermedahl A., Holsti N., Thesing S., Whalley D., Bernat G., Ferdinand C., Heckmann R., Mitra T., Mueller F., Puaut I., Puschner P., Staschulat J., Stenström P., The Worst-case Execution Time Problem – Overview of Methods and Survey of Tools, ACM T EMBED COMPUT S, 2008, 7, 1¸ 45
[8] Henzinger T.A., Kirsch C.M., The Embedded Machine: Predictable, Portable Real-Time Code, ACM T PROGR LANG SYS, 2007, 29

[9] Kopetz H., The Rationale for Time-Triggered Ethernet, Proceedings of the 29th IEEE Real-Time Systems Symposium (30 November - 3 December 2008, Barcelona, Spain) IEEE Computer Society, 2008, 3-11
[10] Astrom K.J., Wittenmark B., Adaptive Control, 2nd edition, Addison-Wesley Longman Publishing Co., Boston, MA, USA, 1994
[11] Gray J., What Next? A Dozen Information-Technology Research Goals, J ACM, 2003, 50, 41-57
[12] Sangiovanni-Vincentelli A., Quo Vadis, SLD? Reasoning About the Trends and Challenges of System Level Design, P IEEE, 2007, 95, 467-506
[13] Caspi P. et al., Guidelines for a graduate curriculum on embedded software and systems, ACM T EMBED COMPUT S, 2005, 4, 587-611
[14] Lee E., Absolutely positively on time: what would it take?, COMPUTER, 2005, 38, 85-87