Proc. Estonian Acad. Sci. Eng., 2005, **11**, 1, 3–17

Challenges for real-time systems engineering. Part 1: State of the art

Leo Motus[a], Robertus A. Vingerhoeds[b] and Merik Meriste[c]

[a] Department of Computer Control, Tallinn University of Technology, Ehitajate tee 5, 19086 Tallinn, Estonia; leo.motus@dcc.ttu.ee
[b] Ecole Nationale d'Ingénieurs, Av. d'Azereix, 65016 Tarbes, France; eaai@wanadoo.fr
[c] Institute of Technology, University of Tartu, 50090 Tartu, Estonia; merik.meriste@ut.ee

Received 26 March 2003

Abstract. This two-part paper addresses characteristic properties of time-critical software-intensive systems (e.g. embedded, real-time, remote monitoring and diagnosis systems). These properties lie in the interface between systems engineering and software engineering. The first part of the paper pinpoints existing problems, the second one discusses potential resolution routes. The first part starts from observations of the current systems development process with the focus on stages for specification of requirements and design. The key problem, in addition to the increase of the overall complexity of systems, is the increasing role of emergent (e.g. non-predictable) system behaviour, its causes and methods for maintaining better control over the system emergent behaviour.

Key words: real-time systems, time-critical systems, proactive components, interactive computing.

1. OBSERVATIONS ON ADVANCES IN SYSTEMS DEVELOPMENT

It has been widely accepted that an increasing number of contemporary societal, business and defence infrastructures, instrumental devices, household appliances and technological processes are essentially software intensive. Many new materials and products can be produced only due to software-intensive technologies. The borderline between hard- and software implementation is dispersing. There are evidences that disciplines, evolving rather independently for several decades, e.g. computer science, artificial intelligence, control theory, systems engineering and software engineering, are today being mixed to better satisfy the system design goals. This is an interesting period where the hitherto separately involved theoretical disciplines must evolve jointly to explain the phenomena observed and solve problems, detected empirically in computer

applications where the computer directly and persistently interacts with its environment.

The start of the new millennium has already given rise to a number of progress evaluations in several domains. The use of digital computers in real-time control has already been addressed in several review papers [1–5]. Still, some fundamental advances and changes in systems characteristics and development process deserve more attention.

First of all, one should note the *strongly increasing complexity of the systems* that are being developed and used. Boeing 777, for example, has over one thousand embedded processors and over four million lines of software to control subsystems and to aid pilots in flight management. Each of these subsystems has a specific task. Some subsystems are present in multiple copies to provide reliability through redundancy (whereas the synchronization between those redundant subsystems poses interesting theoretical and practical problems). All the subsystems are working together, both on board of the aircraft and outside it; for example, interacting with flight control and guidance systems, etc. Such a complexity has developed gradually as the requirements to computer systems have grown in numbers and strictness due to the persistently increasing number of functions that are trusted to computers. This complexity requires specific attention during the system development process, so as to ensure that all requirements are really met and that no side effects will occur. It is therefore necessary to guide the design of these complex systems with a solid, coordinated design approach that allows for a proper verification and validation process. Such a complexity needs specific attention also during the operation of the system – automatic diagnosis, potential dynamic reconfiguration of the system, etc. – that must include some elements of on-line verification and validation to function properly. Less technical examples of complex systems come from the domain of computational modelling, where computer systems and their software is used as a model of a natural phenomenon, e.g. the model for fission reaction, models used in material design, the physiological model of a human, etc.

Then, there is a rising interest in *systems-on-chip* devices, where several parts of a system are combined on one chip. These chips are then integrated into other systems in an embedded manner in order to fulfil several subtasks. A good example is a full-blown self-organizing map neural network with dedicated processing management and DSP front end, all on one chip. This type of devices is now targeting the industrial market and there is an evident need to support the design of the architecture of such devices (e.g. to specify functionality of individual blocks of a system-on-chip, to define relations and interactions of such blocks, to specify the outside relations of a system-on-chip) and the support for designing systems using such components. For instance, methods and tools, which support the integration of systems-on-chip into larger (embedded) systems and help to ensure proper data communication with other systems and with their environment, are still in their infancy. History has already shown negative examples where very efficient data acquisition means were available in such a

chip, very efficient computing power was present, but no means were provided to transfer the acquired and pre-processed data efficiently from the chip to the advanced processing part of the system. In such cases the system did not reach its target, additional external components were to be included, and integrated at a huge cost to fulfil the system requirements.

The abovementioned problems arose, at least partially, because the used design tools and the underlying theory (conventional algorithm-centred model of computation) were not appropriate for embedded applications, where the stream processing is required. Guaranteed success of system design from systems-on-chips and other (pro-)active components assumes the application of interaction-centred models of computation that would better support the analysis of (multiple-simultaneous) stream processing.

The development of *embedded real-time systems* and the corresponding need for a coordinated design, verification and validation are extremely important for industry, and indeed for society, which determines the safety criteria for such applications. There is an evident need for formal analysis in the development of real-time systems, although computer science still does not provide adequate tools that support formal analysis of embedded systems (see the examples given by Tennenhouse in [6]). In many cases when the computing power is insufficient for a given application, mere moving to more powerful processors will not improve the situation. Besides, in an industrial environment the use of more powerful processors is frequently not acceptable for economical or technical (e.g. excessive heat emission) reasons. In addition, there is a natural tendency that the user asks for more, and more complex, tasks to be solved once a more powerful processor is available. The problem will therefore eventually come back.

The need for a coordinated approach in the development of embedded real-time systems, that is based on considering the entire surrounding infrastructure (communications, smart devices, computers at distance, etc.), is therefore evident. Evaluating the state-of-the-art techniques developed over the last ten years, the limitations imposed by the existing underlying theory and available tools for designing and analysing the applications at hand become quickly apparent. This statement is also supported by increasing popularity of interaction-centred models of computation (that include embedded systems as a special case) in computer science and software engineering, as demonstrated in [7]. The expressive and analysing power of those models explain the success of object-oriented design and programming, and enable timing analysis of interactions, among many other things. Before such techniques are mature enough to be introduced to industry, substantial research still needs to be done.

One can also observe increasing use of the so-called *intelligent techniques* within different real-time applications. These techniques, grouped under the common name of artificial intelligence (AI) techniques, allow for the introduction of reasoning approaches that are complementary to hitherto applied algorithmic approaches. High-end developer tools target the market by allowing to develop expert systems to be applied in time-critical tasks, using high-level

knowledge, such as operator knowledge, or neural networks based on learning sets of data [8]. Current state of the art of AI tools allows for specification of knowledge (rules, cases, input/output data pairs, etc.), but hardly ever touch on the systems design and actual software engineering processes. Algorithms have been developed for many important problems in artificial intelligence, such as search algorithms, automatic and interactive theorem proving systems and others. In knowledge engineering, a similar state of maturity has not been reached yet.

Still, practical cases of using agent-based reasoning, approximate, any-time and time-constraint reasoning about knowledge, and the related concepts of *intelligent interaction* and ontology of components are the signs of coming changes. The shift in the character of practical applications has caused the move of AI researchers from algorithm-centred approaches to the agent-based (i.e. interaction-centred) ones that are more sophisticated. This is in accordance with the evolution of software engineering and computer science, from algorithm-centred programming languages and models of computation to object-oriented programming languages and interaction-centred models of computation. The new approach also supports handling of components with incomplete information about their own properties and about the properties of their interaction partners. At the same time this evolution leads to new problems, such as the necessity to use higher-order logic [9,10] and introduction of the time awareness into models and methods [11]. The emergence of new models of computation has pointed out the necessity of time modelling and philosophy-related digital problems, discussed in [12–14], that should not be overlooked.

Knowledge engineering becomes even more complicated when realizing that knowledge can almost never completely be modelled using only one representation technique. The simultaneous use of *multiple-knowledge representation techniques* deserves attention. Information, needed in automation, comes from different sources and is available in symbolic or numerical format, each source covering a part of the current state. For each type of knowledge an appropriate set of representation techniques, matching its characteristics, should be found.

However, this brings forward an additional problem for systems engineering – many enabling theories (e.g. AI and control theory) focus on rather small parts of a system, leaving integration problems of multiple theories with their multiple and accumulating approximations and multiple views on information representations to systems and software engineers. Partial support to resolving this situation comes from the Unified Modelling Language (UML) paradigm, developed as a design tool in software engineering and increasingly used more widely [14].

Finally, it should be noted that contemporary computer systems rarely function in isolation; instead they are almost always collaborating with their environment, interacting with other systems (comprising sensors, actuators, hardware, software, human operators, natural processes, etc.). During the design process of a system, at a certain moment the *integration of different subsystems* needs to be addressed (component-based design, reuse of software and COTS-based design are typical examples that lead to the analysis of integration problems) as

discussed in [15,16]. This entails "matching" the characteristics of the system under development with the characteristics of the collaborating and interacting systems, making sure that the integral behaviour of cooperating systems will adhere to the requirements and will meet all the given constraints, etc.

It should be noted that many design and run-time problems in complex systems actually stem from incomplete match of properties of the interacting partners or are found at the interfaces between different subsystems or components. Another example of malicious and difficult to detect subset of integration problems stem from not quite coherent theories and their approximations as being used in different interacting components.

There is an evident need for a coordinated, structured design approach that takes into account the variety of integration issues in a clear and transparent manner, and that allows for analysing the impact of alternative design choices. One should also address the rapidly increasing autonomy, intelligence, selfishness, and proactive behaviour of system components as described, for instance, in [17]; that may substantially influence system architecture and is an explicit source of incomplete information in a well-designed system. Incomplete information in a system will add to emergent behaviour that results form interactions between autonomous and proactive components of the system during its operation, and cannot be deduced from the given finite design description of the system.

In a nutshell, the new software-intensive systems can be characterized, based on the observations discussed above, by the following features:

- increasing number of new products, systems, devices, and their components (building blocks) are software-intensive, meaning that their functionality is essentially determined by software;
- as a rule, the software is directly interacting with (i.e. monitoring and influencing) the environment, this emphasizes the role of stream processing as opposed to conventional string processing;
- remarkable increase in the complexity of such systems, including intrinsic complexity, can be observed – for instance, the increasing role of emergent behaviour that is generated dynamically and can not be deduced from the static description of the system;
- the interaction of (autonomous, proactive) components has a major role in determining the systems behaviour, reducing correspondingly the role of separate algorithms in the overall behavioural pattern of systems;
- there are clear indications that modelling techniques emerging in software engineering, systems engineering and knowledge engineering focus on integration of multiple views and techniques;
- the research focus is moving from research of isolated problem solving algorithms to integration of algorithms into a system so that prefixed system properties will be achieved as precisely and economically as technically possible.

The above-mentioned issues that represent the essence of evolution trends observable in software-intensive systems as built for embedded and time-critical

applications (but also for many proactive, autonomous and pervasive computing systems) form the basis for identification of perspective research issues, discussed further in this paper.

2. TASKS FOR A SYSTEM DESIGN ENVIRONMENT

There is a strong need for tools that would provide the designer of industrial systems with overall theoretical and practical support in system and software engineering. Bits and pieces of this approach already exist and are applicable at several stages of system development. However, the overall framework, that allows for proper support during the multiple stages of the system development cycle, e.g. specification, analysis, design, implementation, testing, and at the same time would promote representation and comparative analysis of different architectures and components, is still missing. Such a framework should be concerned with large (potentially embedded) computing systems with an objective to handle all the major aspects of system development – starting from the initial idea of a system up to the fully described, verified, implemented and validated system that meets the user requirements, including the quality of service (QoS) requirements imposed on fault-tolerance, safety, security, etc. This development framework should satisfy the features of the contemporary software-intensive systems listed in the previous section of this paper.

Starting questions should focus on *how to handle the ever-increasing complexity*, on the analysing methods that enable better understanding of the reasons that influence system behaviour, and better prediction of the actual behaviour. This involves several aspects of system representation, taking into account both functional aspects as well as QoS aspects (such as timing correctness, fault-tolerance, safety, security, etc.). Quite often, QoS requirements can be satisfied only if one can analyse and control the intensity of the emergent behaviour.

System description should also include the processing of *location-dependent aspects*, concerning the manner in which the physical or logical locations of components within a complete system are described and how the connections of the components depend on their locations. In many cases such details may have an impact on the system behaviour. For instance, consider the impact of a component failure at a given location (e.g. a wagon of a train) that causes the occurrence of a symptom elsewhere (e.g. in another wagon), or interactions of autonomously moving vehicles. This type of information has been shown to be of vital importance in the success of system development in the project BRIDGE [18]. Good progress has been achieved already in the analysis of the logical behaviour of systems, but for the formal analysis of time and location aspects the progress is not as good.

The system designer faces a constant need for compromises by selection of tasks to be performed, their implementation either in hardware or in software,

selection of communication media to be used, selection of methods for the assessment of the progress in system development, etc. The compromises may be affected by technical or economical limitations or even by subjective preferences of the endusers of the designed system. It might be that a choice is made for digital signal processing techniques instead of a hardware component (usually based on analogue processing), or an ASIC chip is used instead of a piece of software-implemented production rules in an expert system, etc. This means that particular *architectural (and algorithmic) choices will be finally fixed only during the system design and testing process*. It should be noted here that the design engineer is frequently left without any supporting tool that would help to forecast, early at the design stage, the realistic impact of the alternative design choices on the behaviour of the system.

An additional factor, often neglected in academic studies but very important in practice, concerns the strictly limited development time, assigned for many industrial projects. Sometimes, over a limited period of time, a high number of software packages with still evolving contents are delivered to the customer, frequently on still evolving hardware platforms. This imposes a seriously reduced time frame for the design, development and validation of the software, as well as its integration with surrounding systems. Within this framework, under the time pressure a good supporting design environment has an even greater value. There is some light at the end of the tunnel – new evolving system development technologies based on UML, for instance [19,20], are promising.

The industry is aware of the necessity to introduce proper system and software development approaches. This is underlined by the eagerness to obtain and show Capability Maturity Model (CMM) levels, and other measures for maturity in system development processes, requirements traceability, etc. Many companies that were between levels 1 (initiation) and 2 (defined at project level) have shifted during the last few years to the levels 2 and 3 (defined at company level). This is a clear illustration of industry's adherence to system and software development processes. Another illustration is the strong support to Object Management Group (OMG) and wide acceptance of UML, the related software products (e.g. [14]), and a software process adapted to the new technology, developed by this consortium for the industrial use.

Software engineering attempts hard to move software production process closer to the reliability and predictability level, achieved in other engineering disciplines (e.g. in civil and mechanical engineering). Whereas progress has been quite smooth in data and information processing software, the embedded software and other software-intensive products have suffered difficulties. Ironically, the major obstacle to the progress has been the incoherence between the theory (algorithm-centred mainstream of computer science in the last century), essentially different computing process in new applications (stream-based instead of string-based computing), and the increasingly component-based practice of software production. Only in 1990s the new paradigm (interaction-centred computation) has gained wider acceptance and, at least in principle, coherence

between the solved tasks, goals to be achieved, applied theory, and available tools have started to develop. Today, the theoretical foundations for the new paradigm are still developing [9,21] and are not yet sufficiently mature to be applied in building industrial systems and software engineering tools for industrial use. The experimental applications however, have been promising – a discussion of this topic is presented in the second part of this paper.

Unfortunately, over the same time, the problems to be solved and the architectures to be handled have been getting more complex as well, asking for well-elaborated approaches. Some of the new problems were characterized earlier in this section. Somewhat different aspects of new computer applications are emphasized in [6] by discussing the "movement from human-centred computing to human-supervised (or even unsupervised) computing". Techniques and tools that are being used for system engineering, have an effect not only on the problem-solving ability of a designer, but also on avoidance and detection of errors that are made during problem solving. Thus it is clear that these tools and methods should also cater for human limitations and capabilities. These issues are referred to as *cognitive engineering*, which term combines ideas from system engineering, cognitive psychology, and other human factors related research.

Many of the arising problems originate from the insufficient integration of software engineering, system engineering and cognitive engineering in the building process of complex systems. The problems often arise in the interfaces between the components, where the components may be of hardware, software, environment, or human origin. These problems are often caused by incompletely matched ontology of the interacting components. This all leads to the search of methodologies that would foster coordinated design of the components, interfaces and interactions between the components, and that would provide seamless transitions and mappings between the disciplines involved. Emerging theoretical and technological trends, such as interaction-centred models of computation, agent-based reasoning methods, proactive computations and ontology of interactions also support the aforementioned ideas. In addition to theoretical research, several initiatives on new computing technologies have been launched recently (for instance, proactive computing [6,22] and autonomic computing [23]).

Looking at software-intensive systems from an overall systems point of view, every aspect and part of the overall system has to be assessed for correct functioning and reliability; e.g. the process to be controlled must be understood with sufficient precision, equipment used to monitor and influence the controlled process must be correctly interfaced with the environment and with the computer system and dynamic properties of the software must be matched with those of the environment and the interfacing equipment. All systems will eventually have failures, either on system or on component level, related to the technological limits and the limits of the knowledge about the system. This means that theories, technologies, and the tools used for system development and pre-implementation analysis of the system design must support all the aforementioned aspects. New theories and technologies are emerging but, at the moment, there are no tools of

the required analysis power available for the industry. Majority of the available tools are still based on models of computation (e.g. Turing machines with slight extensions [21]) that have insufficient expressive and analysis power for handling new software-intensive systems.

Note that these observations apply to a broader scale than just to embedded real-time systems. Software-intensive systems development for operation in a "non-real-time environment" needs to resolve very similar multidisciplinary design problems, compromise of the resources to be applied for a given task (software, hardware or human) and validation and verification in safety-critical applications. Although system safety engineering techniques have existed for decades, changes and extensions are required in techniques used for systems that contain digital computers and software. System design should therefore comprise topics such as modelling and analysis of safety, system and software requirements specification, safe software design, software fault tolerance, and verification and validation of safety. Safety topics should be seamlessly integrated into the research of system architecture. In addition to long-known safety problems, contemporary systems often fail in security issues. The security issues have become as important as safety issues and need remarkably more attention in system design.

This section focused on the designer's expectations with respect to integrated systems design environment. The majority of software-intensive systems comprise highly multidisciplinary knowledge and corresponding theories. The designers of such systems often face unexpected behaviour of the system due to cumulative approximation errors invoked by integrated use of many approximately implemented theories and methods, e.g. loss of stability in a control algorithm due to excessive jitter of time unit (as used by a discrete time theory based algorithm). Commercially available design environments are pretty helpless in analysing such phenomena. Designers have a serious need to get automated support in comparing the effects of alternative design decisions and compromises on the system behaviour. Finally, computing-related part of the system design support suffers from an obvious mismatch between the available theory of computing and the essence of the problem to be solved.

3. NEED FOR NEW MODELS OF COMPUTATION

Looking at the essence of modern computer control systems, communication systems, contemporary multi-media applications, distributed (artificial intelligence) systems and others, one cannot overlook the influence of time constraints imposed upon their behaviour. There are two major causes for the introduction of time constraints [24]:

– necessity to match the behaviour of a computer system with that of its independently functioning interaction partners in the environment, whose behaviour is to be influenced by the computer system;

– to represent the incompletely known, or too sophisticated, causal relations approximately; for instance, those that determine invocation of the system components, and synchronization of various interacting activities between the partners.

As a consequence, the designer has to deal with the construction of a time-constraint concurrent software system that may have safety- and time-critical and fault-tolerant features that need verification. As a precondition to verification of time-constraint software one needs time-sensitive models of computation. It has been explained earlier in this paper that time-sensitive models of computation should, at the same time, be preferably also interaction-centred and not algorithm-centred, as they conventionally are.

It should be noted that conventional theories of computer science and tools in software engineering are not able to address the full suite of timing issues, as the role of time has been reduced to minimum in those theories and tools since the canonization of the algorithm theory as the basis for computer science in the 1960s [25]. Therefore the timing issues in software-intensive systems and in common software have been addressed, in the majority of cases, only partially – up to the point that is possible in the conventional algorithm theory. This means that performance and scheduling issues have been addressed, whereas timing of interactions and validity intervals for events and data have been usually neglected.

Rapidly increasing complexity of software-intensive systems, growing popularity of building systems from (autonomously and proactively functioning) components, wide acceptance of the object-oriented design methodology [14] together with the increasing role of safety, security, and time-awareness issues and improved theoretical understanding of the essence of computing, has given rise to studies in interaction-centred models of computing and in the corresponding system design paradigm. In fact, the first attempt to emphasize the role of interactions in computation can be dated back to the end of 1930s, e.g. Turing's choice machine [26]. The second attempt started by end of the 1970s; a summary of five-year developments in interaction-centred models of computation is published in [27] and the first time-sensitive, empirical, interaction-centred model is published in [28].

3.1. Interaction-centred models of computation

Component-based architectures, object-oriented design and implementation methods, increasing autonomy and pro-activeness of components are the basic sources of difficulties hindering the application of algorithm-centred models of computation for the analysis of system properties. Actually, the difficulties are caused by an attempt to apply prescriptive specification in the case where only behavioural specification is applicable. The above-listed methods and technologies enable explicit handling of incompletely known causal relations. The designer of such systems cannot avoid the presence of the incompletely known inner structure and pro-activity invoked properties of components, externally

invoked methods in objects and dependence of the components behaviour on the history of computations. Quite often components exhibit indefinitely ongoing behaviour, not necessarily strictly coordinated with that of the other components they interact with. In many cases, ongoing behaviour can be considered as a cyclic or sporadic repetition for indefinitely many times of a finite set of algorithmically described activities. Interactions between the components with ongoing behaviour lead in many practical cases to the necessity of time-selective interactions as demonstrated in [10].

The connections between the components in a system are usually static; however, the connections do not describe unambiguously the behaviour of systems. The system behaviour depends, to a large extent, on the actual semantic contents of messages exchanged during a particular interaction act performed between connected components during the operation of the system. Such systems exhibit the so-called *emergent behaviour*, i.e. a behaviour that cannot be deduced from the description of the static structure of the system. There can often be a countable number of emergent behaviours; therefore verification of behavioural properties of such systems needs innovative approach as compared to methods applied for conventional programs, whose behaviours can be deduced from the prescribed static structure of the program.

Several authors have suggested that serious extensions to the existing models of computation are necessary so as to address explicitly interactions, emergent behaviour and more subtle timing constraints (imposed upon the interactions). These extended models should exceed conventional algorithm-centred models both in formal description power and analysis power. As a rule, those models describe indefinitely ongoing computations on interaction machines as defined in [9] instead of using the conventional Turing machine. Interaction machine can be either sequential or multistream interaction machine. Two Turing machines that jointly process a data stream can model a simple sequential interaction machine (called the persistent Turing machine), if those Turing machines, while processing the stream, are communicating via an independently operating proactive device (e.g. an active filter) with memory. This means that the input of the second Turing machine may depend on the pre-history of computations on the first Turing machine. The notion “Universal Turing Machine”, introduced by Chaitin in [12], is intuitively close to the multistream interaction machine notion as introduced in [9].

Interactive computing can be viewed as agent-based processing of streams, as opposed to the algorithmic computing where an input string (i.e. an element of the input stream) is processed. The output string (an element of the output stream) is generated and then the algorithm terminates with no memories of the past. The input/output stream processing emphasizes the persistent nature of computing. Characteristic to interactive computing is the phenomenon of emergent behaviour. The following features are typically present in interaction-centred computations and are absent from algorithm-centred computations:

– (multiple) stream input-output for a computing agent;

– interleaving of inputs and outputs during computations is natural;
– history-dependent behaviour of the computing agents.

Processes that exhibit above-listed properties are hard to describe and even harder to analyse in algorithm-centred models (for instance, learning and adaptation in a system) and can naturally be presented in interaction-centred models.

Academic research of interaction-centred models of computation has been quite intensive. For instance, Milner developed a π-calculus [29,30] that extends his interaction-centred model [27] to mobile, autonomous computing agents. Wegner [31] clarified the philosophical background of interaction-centred computing. Goldin and colleagues [32] demonstrated that the modelling power of UML descriptions corresponds to that of interaction machines. The essence and practical presentation aspects of interaction machines has been clarified by many authors [9,33–36]; however, the area of interaction machines is still open to research.

So far the research in this area has been mostly focused on theoretical-philosophical aspects of interaction-centred computing and has not yet led to tools that perform formal analysis on systems described in terms of interaction machines. Necessity to accelerate progress in formal analysis is soon moving to the focus of system and software engineering. This is partially caused by OMG that launched a large-scale project, focused on developing a new methodology (see http://www.omg.org/mda/) for designing software-intensive systems that assumes the availability of formal verification at the specification and design stages. In addition, most of the computer applications in industry need remarkably wider support from formal analysis – the complexity of those applications demands combining of testing with formal methods to reach the required level of QoS.

A conventional computing system is based on algorithms and completely known causal relations between algorithms. Those causal relations determine admissible permutations in interleaving input and output streams (i.e. admissible concurrency of computations). When some causal relations are not completely known, the designer cannot precisely know which of the occurring permutations are admissible. This difficulty is directly related to permutations, potentially occurring when a system performs forced concurrency. The forced concurrency is a normal operation mode of multi-stream interaction machines, when physical processes in the environment (and not the designer, or available number of processors) determine the number of concurrent computing processes, called forced concurrency in [10] and true concurrency in [9].

To verify, in the general case, the forced concurrent mode of processing in software-intensive systems, one needs a separate time-counting system for each stream – this causes introduction of the sophisticated time model [13] to the system description. Similarly, the corresponding interaction-centred model of computation has to be time aware. For instance, time-selective exchange of information between concurrently processed streams, as a side effect of forced concurrency leads to checking the validity time of transmitted information. Time-

correctness of the behaviour of such systems cannot be demonstrated by testing only. One also needs formal verification enabled by a time-aware, interaction-centred model of computation [10].

4. CONCLUSIONS

Rapidly increasing number of systems and commodity products essentially depends on computers and software (so-called software-intensive systems). This means that the functionality of those systems and products is actually determined by software. Part 1 of the paper surveyed the basic characteristics of such systems, deduced the basic new requirements to the system architecture and software, and assessed the available theory, technology and methods for building such systems. Based on those characteristics, requirements, and assessment the paper sketched main evolution trends in the underlying theory of computation and technologies used in systems and software engineering.

In a nutshell, theory of computation moves from the algorithm-centred paradigm to the interaction-centred paradigm, systems and software engineering is focusing on problems related to increased use of proactive building blocks. The mentioned trends are strongly influenced by interdisciplinary efforts to create proactive (software) components, enable their collaboration in a system, and guarantee that the system satisfies the expectations of its users. Part 2 of this paper focuses on a survey and discussions on those interdisciplinary efforts in developing and testing new theories and technologies to support the designers of such systems.

REFERENCES

1. Holt, J. D. Current practice in software engineering, a survey. *Comput. Control Eng. J.*, 1997, **8**, 167–172.
2. Cervin, A., Henriksson, D., Lincoln, B., Eker, J. and Årzén, K.-E. How does control timing affect performance? *IEEE Control Syst. Mag.*, 2003, **23**, 16–30.
3. Jennings, N. R. and Bussmann, S. Agent-based control systems. *IEEE Control Syst. Mag.*, 2003, **23**, 61–73.
4. Sanz, R. and Zalewski, J. Pattern-based control systems engineering. *IEEE Control Syst. Mag.*, 2003, **23**, 43–60.
5. Selic, B. and Motus, L. Using models in real-time software design. *IEEE Control Syst. Mag.*, 2003, **23**, 31–42.
6. Tennenhouse, D. Proactive computing. *Commun. ACM*, 2000, **43**, 43–50.
7. Meriste, M. and Motus, L. On models for time-sensitive interactive computing. *Lecture Notes in Comput. Sci.*, 2002, **2329**, 156–165.
8. Vingerhoeds, R. A. *Génie d'automatisation et systèmes intélligents temps réels, Proposition d'une méthodologie de conception.* Dossier d'habilitation a diriger des rechèrches, Institut National Polytechnique Toulouse, 2002.
9. Wegner, P. Interactive foundations of computing. *Theor. Comput. Sci.*, 1998, **192**, 315–351.
10. Motus, L. and Rodd, M. G. *Timing Analysis of Real-time Software.* Pergamon/Elsevier, 1994.

11. Motus, L. and Meriste, M. Towards self-organising time-sensitive control system's software. In *Proc. IFAC Conference on New Technologies in Computer Control*. Hong Kong, 2001, 236–241.
12. Chaitin, G. J. Meta-mathematics and the foundations of mathematics. *Bull. Eur. Assoc. Theor. Comput. Sci.*, 2002, **77**, 167–179.
13. Motus, L. Modeling metric time. In *UML for Real: Design of Embedded Real-time Systems* (Selic, B., Lavagno, L. and Martin, G., eds.). Kluwer, Norwell, 2003, 205–220.
14. Object Management Group. *UML Profile for Schedulability, Performance, and Time: Specification*. OMG document ptc/2002-03-02, Needham, 2002.
15. Kobryn, C. Modeling components and frameworks with UML. *Comm. ACM*, 2000, **43**, 31–38.
16. Sparling, M. Lessons learned through six years of component-based development. *Comm. ACM*, 2000, **43**, 47–53.
17. Ferber, J. *Multi-Agent Systems*. Addison-Wesley, Harlow, 1999.
18. Coen, L. De, Netten, B. D. and Vingerhoeds, R. A. Central advice system for fleet management and operations; improving the safety and reliability for rolling stock. In *Second World Congress "Safety of Transportation; Imbalance Between Growth and Safety?"*. Delft, 1998, 1–10.
19. Selic, B., Lavagno, L. and Martin, G. (eds.). *UML for Real: Design of Embedded Real-time Systems*. Kluwer, Norwell, 2003.
20. Mellor, S. J., Kendall, S., Uhl, A. and Weise, D. *MDA Distilled*. Addison-Wesley, Boston, 2004.
21. Blass, A. and Gurevich, Y. Algorithms: a quest for absolute definitions. *Bull. Eur. Assoc. Theor. Comput. Sci.*, 2003, **81**, 1–30.
22. Hamilton, S. Intel research expands Moore's law. *IEEE Computer*, 2003, **36**, 31–40.
23. Kephart, J. O. and Chess, D. M. The vision of autonomic computing. *IEEE Computer*, 2003, **36**, 41–50.
24. Motus, L. and Meriste, M. Time modelling for requirements and specification analysis. In *Real-time Programming 2003* (Colnaric, M., Adamski, M. and Wegrzyn, M., eds.). Elsevier Science, Oxford, 2003, 9–14.
25. Lee, E. A. What is ahead for embedded software? *IEEE Computer*, 2000, **33**, 18–26.
26. Turing, A. On computable numbers with an application to the Entscheidungsproblem. *Proc. London Math. Soc.*, 1936, **2**, 230–265. Corrections: 1937, **3**, 544–546.
27. Milner, R. A. A calculus of communicating systems. *Lecture Notes in Comput. Sci.*, 1980, **92**.
28. Quirk, W. and Gilbert, R. *The Formal Specification of the Requirements of Complex Real-time Systems*. AERE, Harwell, Rep. No. 8602, 1977.
29. Milner, R. *Communicating and Mobile Systems: The PI-calculus*. Cambridge Univ. Pr., 1999.
30. Milner, R. A. Elements of interaction. Turing Award lecture. *Comm. ACM*, 1993, **36**, 78–89.
31. Wegner, P. Why interaction is more powerful than algorithms. *Comm. ACM*, 1997, **40**, 80–91.
32. Goldin, D., Keil, D. and Wegner, P. An interactive viewpoint on the role of UML. In *Unified Modeling Language: Systems Analysis, Design, and Development Issues* (Siau, K. and Halpin, T., eds.). Idea Group Publ., Hershey, PA, 2001, 250–264.
33. Meriste, M. and Penjam, J. Attributed models of computing. *Proc. Estonian Acad. Sci. Eng.*, 1995, **1**, 139–157.
34. Gurevich, Y. Evolving algebras. In *Proc. 13th IFIP Congress*. Hamburg, 1994, vol. 1, 423–424.
35. Gurevich, Y. and Spielmann, M. Recursive abstract state machines. *J. Universal Comput. Sci.*, 1997, **3**, 233–246.
36. Motus, L. Timing problems and their handling at system integration. In *Artificial Intelligence in Industrial Decision Making, Control and Automation* (Tsafestas, S. G. and Verbruggen, H. B., eds.). Kluwer, 1995, 67–88.

Sardsüsteemide arendustehnoloogia kitsaskohad. 1. osa: Hetkeseis ja üldtrendid

Leo Mõtus, Robertus A. Vingerhoeds ja Merik Meriste

Selles kaheosalises artiklis on analüüsitud ajakriitiliste tarkvaramahukate süsteemide võtmeprobleeme, mis paiknevad kasutatava süsteemitehnika ja tarkvaratehnika ülekattealal. Taoliste süsteemide rakenduste näideteks on arvuti poolt juhitavad liikuvad seadmed, tehnoloogiliste protsesside juhtimissüsteemid, meditsiiniaparatuur, robotid, koduelektroonika, mobiiltelefonid, looduskeskkonnaseire jms. Artikli esimeses osas on esile toodud teoreetilised ja tehnoloogilised kitsaskohad. Käsitlus baseerub eksisteerivate süsteemide analüüsil, milles eriline tähelepanu on suunatud spetsifitseerimis- ja projekteerimisetappidele. Uus raskelt analüüsitav omadus, mis iseloomustab tarkvaramahukaid süsteeme, on ilmnev käitumine, mis ei ole ennustatav süsteemi staatilisest kirjeldusest, vaid tekib dünaamiliselt süsteemi töö käigus.