Designing Self-Adaptive Embedded Real-time Software -
Towards System Engineering of Self-Adaptation

Franz Josef Rammig, Stefan Grösbrink, Katharina Stahl, and Yuhong Zhao
University of Paderborn
Heinz Nixdorf Institute
Paderborn, Germany
{franz, s.groesbrink, katharina.stahl, zhao}@upb.de

Abstract— **Upcoming interlinked embedded systems will be confronted with unexpectedly changing environments, which makes online adaptation without manual interference necessary. There is a need for appropriate system architectures and novel design approaches. In this paper, we discuss general concepts of self-adaptive real-time systems. Furthermore, specific system engineering techniques solving two important aspects of such a paradigm are presented. We discuss how the necessity for adaptation can be identified using Online Model Checking and how self-adapting safety guards can be designed by means of Artificial Immune Systems. Finally, we present an approach to integrating these techniques into an underlying platform architecture based on mixed-criticality virtualization.**

Keywords— Cyber Physical Systems, Self-adapting Software, Online Model Checking, Artificial Immune Systems, Danger Theory, System Virtualization, Mixed-criticality Systems

I. INTRODUCTION

Currently a dramatic shift in the nature of embedded software can be observed. For many years, embedded software was characterized by more or less fixed sets of applications, managed by a real-time operating system (RTOS) which is carefully tailored in such a way that it exactly serves the needs of the applications, avoiding as much as possible any redundant service. In the beginning, even changing environments have been neglected. The user did serve as a kind of intelligent filter being responsible to adapt the modified environment to the fixed embedded system. Even when embedded systems were designed in a more adaptive manner, this adaptation took place off-line under control by human experts. However, embedded systems are reactive by nature. This becomes even more evident in the case of Cyber Physical Systems [1], which are closely linked to both, the physical environment and the cyberspace. Such systems are embedded into environments, which are predictable only to a certain degree. To deal with this not fully predicable behavior of the environment, one solution might be to include adequate reactions to each imaginable behavior of the environment into the applications and the operating system. This, of course, may result in an immense waste of resources. If the vast majority of imaginable situations will never occur, all the reactions coded-in to handle these situations are overhead, which cannot be avoided because it is unknown beforehand which situations are those to be handled in reality. The situation becomes even more complicated when the embedded systems are directly interlinked without human interference (i.e. without intelligent interface).

Upcoming systems of systems can be seen as a fabric of embedded systems that are interwoven either statically or dynamically. The situation of dynamically creating or deleting communication links further complicates the situation. The German “Industry 4.0” research program may be used as an example. In this vision, the entire fabrication process is treated as an interlinked system of various production sites, controlled at various levels of abstraction, influenced by numerous dynamic parameters (prices, availabilities, regulations, environmental conditions, … etc.), dynamically making and cancelling contracts with suppliers and customers, being influenced by sophisticated logistics systems. This means that even the detailed control of some local machinery may be influenced by some unexpected parameters originating from some remote location, which never have been expected when this piece of software has been designed. Consequently, we have to deal with embedded systems that must be able to continuously adapt to changing environments, which leads to self-evolving software. From this point of view, self-adapting embedded systems are not some academic theory but a necessity for upcoming highly interlinked systems.

When such self-evolving components form a fabric of self-evolving structure, some emergence of the total system of systems may be the consequence. This implies that engineering techniques have to be developed that allow for carefully designing such systems, for efficiently implementing them, and for continuously supervising them. In this paper, we will concentrate on the aspects of detecting the need for change and on continuously supervising self-adapting systems. We will argue from an operating system point of view. In both of the two discussed aspects, sequences of system calls seem to be the adequate granularity of monitoring system behaviors. In addition, we will present our solution to use system virtualization via a real-time-capable hypervisor as a unified platform for self-adapting embedded real-time software.

The rest of the paper is structured as follows: In Section II some basic principles of self-adapting systems are discussed. In Section III it is described how our Online Model Checking technique can be applied to proactively identify needs to adapt the currently running software system. An AIS-based approach for continuous system supervision is presented in Section IV. Section V deals with real-time-capable virtualization concepts. In Section VI a summary and an outlook are given.

II. Self-Adapting Systems

A. General Concepts of Self-adapting Software

Adaptability means that such a software system may evolve at run-time, coping with changing requirements and information provided by the respective environment. Self-adaptive systems are becoming of increasing interest in research. It seems to be natural to use closed-loop control as the basic paradigm to address the aspect of self-adaptation. A well-known model for applying closed-loop control to software is the *MAPE* architecture [2]. Here "***M***" stands for "Monitor" (the current behavior), "***A***" for "Analyze" (the monitored behavior), "***P***" for "Plan" (updates), and "***E***" for "Execute" (the planned updates). A couple of researchers concentrated on this closed-loop point of view for self-adapting software. For an excellent overview about the relevant literature, see [3]. Dealing with real-time systems, however makes it necessary to refine these concepts, see subsection II.*B*.

We are restricting ourselves to software systems where the entire software environment is composed of well-defined components with highly standardized interfaces. We assume that the considered embedded systems are permanently connected to the Internet, which serves as provider of information and services, including a worldwide market of such components. The adaptation process then means either replacing currently running components or adding components to existing software solutions by components that in both cases are extracted from this virtual market place.

In [4] we have identified four prerequisites for solutions where a fully automatic environment is the intent:

(1) First of all, a continuously and automatically accessible market of software components, which follow some strict standards, must exist. Within the scope of this paper, we will not discuss this aspect; we just assume that such a market together with the appropriate access mechanisms exists. Work into this direction is being done, e.g. in the Collaborative Research Center On-the-Fly Computing at the University of Paderborn [5], funded by the German Science Foundation.

(2) Obviously, a technique is needed that allows for identifying the necessity of adaptation. In [6], Carlo Ghezzi and his coauthors argue that online verification might be a good approach for this purpose. If a model of the respective environment is part of the entire model of the considered system, online verification can identify situations where the observed real environment no longer matches its model. In Section III we will discuss how we are using our own Online Model Checking mechanism based on Bounded Model Checking with sliding initial states [7] for this purpose.

(3) Whenever the necessity of an adaptation has been identified it has to be carried out. We are coping with real-time systems. Therefore, we are working on a fully automatic adaptation approach being integrated into a real-time operating system as underlying engine.

Assuming a strict component-oriented software environment, any adaptation means deletion, insertion, or replacement of components. Furthermore, we assume that for components there may exist various profiles. All profiles of a component share the same principal functionality, however under different demands of resources and providing different quality levels of the principal functionality. Usually less resource-demanding profiles provide some lower quality. In a real-time system, any modification of the set of applications to be executed means to perform an acceptance test. If this acceptance test fails, the entire set of components has to be re-arranged in such a manner that the resulting task set will become feasible. In our approach, we make use of the various profiles of the components to find a feasible configuration, which provides maximal overall quality. Algorithmically this is equivalent to solving a Knapsack problem.

(4) In the case of dynamically adapting systems, any kind of change may result in malign operation. Biological organisms therefore feature highly sophisticated protection mechanisms, especially the immune system. Higher organisms like mammals have an immune system, which is adaptable by its own. Comparable concepts are needed for self-adapting software systems as well. Besides using our Online Model Checker not only for triggering adaptations but also for verifying the resulting system [8], we are applying special Artificial Immune Systems (AIS) as adaptive safety guard. We selected a special kind of AIS [9] based on the so-called Danger Theory [10]. This aspect will be discussed in Section IV.

Finally, an adequate architecture for a system platform is needed. We have to deal with primary services, namely the intended behavior of the embedded system and secondary ones. Those include all the services needed for adaptation and supervision. Making use of mixed criticality virtualization seems to be an adequate solution. In Section V we will shortly describe our approach for such a platform.

B. Real-time-aware System Engineering

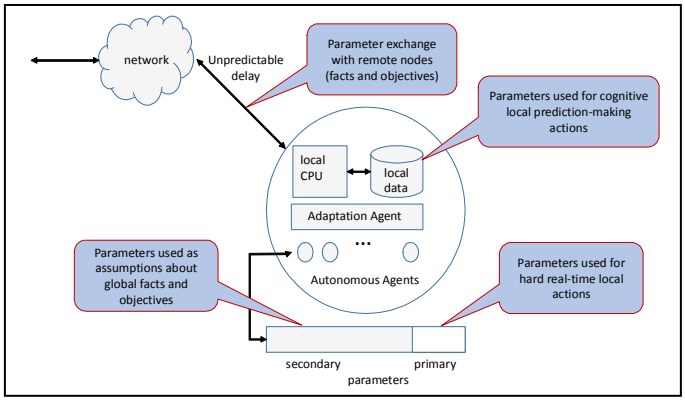

Fig. 1. Global Architecture of Interwoven Self-adapting Systems

We are dealing with highly distributed systems, linked over networks, which are not real time capable (the internet). On the other hand, we are dealing with real-time-critical applications.

As a means to overcome these contradicting properties, we follow a layered approach. We consider a system to be a network of agents, being clustered into distributed computing nodes (see Fig. 1). Such an agent locally acts as an embedded real-time system, providing real time capable services like closed loop applications. For this purpose, it is directly connected to its local sensors and actuators and accesses a set of local variables. Global variables can be accessed only via channels, which are not real time capable by nature. Therefore, each local agent must include some intelligence to deal with assumptions about these global variables to be valid and up-to-date. In addition, an agent must be able to revise its assumptions. Whenever an agent observes a mismatch between its assumptions and the actual data received via the network some adaptation has to take place.

We follow a model-based approach. To enable an adaptation control loop, the respective model also has to include a model of the environment (compare [3]). This consists of the local environment, which is sensed by the local sensors and influenced by the local actuators under real-time constraints (local parameters), but also the entire network to which the local agent is connected (global parameters). When these parameters describing the environment are running out of some predefined constraints, the respective model of the environment is no longer valid and therefore the model of the application may no longer satisfy predefined requirements. If so, some adaptation is needed. Thus, in our system engineering approach, we are following a concept where the construction of software is becoming a highly dynamic on-line process. Instead of creating a fixed release of a software system as the result of an off-line engineering process, a continuously evolving flow of releases is generated. In this sense, we interpret self-adapting systems as self-agile systems. Providing fully automatic on-line agile programming requires to dramatically restricting the supported area of systems and the methods to be applied. In our case, we restrict ourselves to software systems that follow a strict component-based approach: we assume the entire software system to be composed of well-defined components with well-defined and highly standardized interfaces. Any kind of adaptation then means removing, adding, or exchanging such components. Such a coarse-grained granularity results in a substantially reduced scope of potential alternatives. Per computing node, the entire process (primary applications and adaptation process) can be implemented using a mixed-criticality, real-time-capable virtualization system. The primary (hard real-time) applications can be grouped to be executed by a high criticality virtual machine (VM), while the adaptation services run on a lower criticality VM.

At our university, we have extended the adaptation control loop to a two-level one [11]. The lower level, called "Reflective Operator" executes the adaption itself. In our system architecture, it runs on a dedicated (real-time) VM. The upper level, called "Cognitive Operator" takes care of long term strategic planning of adaptions (somehow comparable to the "Plan" activity in the MAPE architecture). It runs under no or soft real-time constraints and may be executed on an even lower criticality VM. In Section V we will discuss especially how to integrate online model checking and AIS-based self-diagnosis into such a virtualized system architecture.

III. IDENTIFICATION OF NEED FOR ADAPTATION

Online Model Checking is in the context of this work not used as a means of verification but to identify situations where the present model of the environment is no longer consistent with the currently sensed values. That is, we assume that the behavior of the component would be still correct with respect to the previous environment, which implies that it is the changing environment, which leads to the violation. In case of a violation, the online model checking mechanism can inform the underlying operating system. The generated counterexample then can be converted into a web query for a problem solving alternative component.

A. Online Model Checking

To apply online model checking, we need to have the model of the component including a model of the environment as well as the property to be checked at hand. The properties of interest in our context are inconsistencies between the modelled environment and the values sensed from the environment that cannot be handled correctly by the actual version of the component. Such a property can be any invariant or non-trivial linear temporal logic formula (i.e., safety or even liveness property). Both the safety checking and liveness-checking problem can be transformed into forward reachability analysis [12] [13]. Online model checking is in essence a type of simplified Bounded Model Checking (BMC) [14] with sliding initial states, applied at runtime.

We require that the current state s_i of the component under test is monitored and stored in a (ring) buffer from time to time during the system execution. The left hand part of Fig. 2 illustrates the basic idea [7] of our online model checking mechanism. Whenever the online model checker is triggered, it tries to take a new (concrete) state, say s_i, from the buffer periodically. It then goes to search for a potential error state in a partial state space of the system model starting from the corresponding abstract state $\hat{s}_i = \sigma(s_i)$, where the function $\sigma(s_i)$ maps each concrete state s_i at the implementation level to the corresponding abstract state $\hat{s}_i$ at the model level. In each checking cycle only finite (transition) steps, say the next k steps, starting from the observed state are scanned. In this way, the state space to be explored is reduced dramatically and the state space explosion problem is avoided. Our goal is to figure out whether there exists an error path starting from some observed state to the unsafe region, i.e. a set of error states. If the property being checked is violated, it implies that the actual reality might not conform to the modelled environment any more, recall that we assume the behavior of the component being correct.

There are many ways to improve the performance of online mode checking (see [15]). Among them, we can speed up online model checking algorithm by reducing the workload of the online model checker. For this purpose, we need to extend the original set F_0 of error states to become $F' = F_0 \vee F_1 \vee \ldots \vee F_n$ by offline backward reachability computation up to n time steps as shown on the right hand part of Fig. 2.

The offline computed unsafe region might also serve to characterize a component in the repository of candidate components. For this purpose, the counterexample generated

by the online model checker in case of a property violation is converted into a query for a component, which does not contain the respective unsafe region.

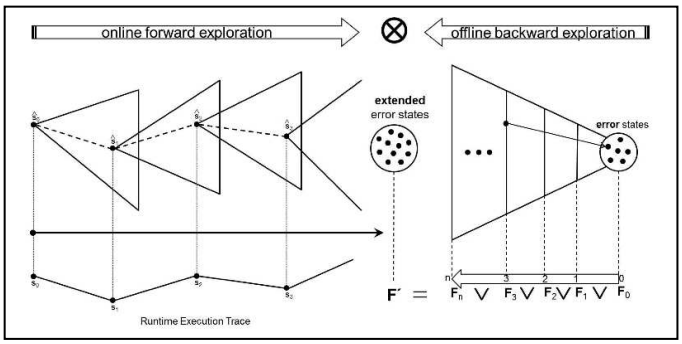

Fig. 2. Online and Offline Process of Online Model Checking

For a more detailed description of our online model checker mechanism, please refer to [8].

B. Observation Granularity

A failure can be malign according to Kopetz [16] only when being passed to the outside. This can happen in an OS-based system only under control of the OS, via a system call. All other failures are benign. As we are considering only OS-based systems, any implementation of an application has to contain a sequence of system calls. Whenever a system call is invoked, we can monitor the state information of the implementation and its actual environment. Therefore, the sequence of system calls of an application is the appropriate level of granularity. As system calls happen anyhow during the system execution, online verification can be integrated as part of the system call handler of an RTOS, by this causing no additional context switch overhead and the necessary information being available without crossing address space borders. In this sense online model checking becomes an RTOS service as shown in Fig. 3.

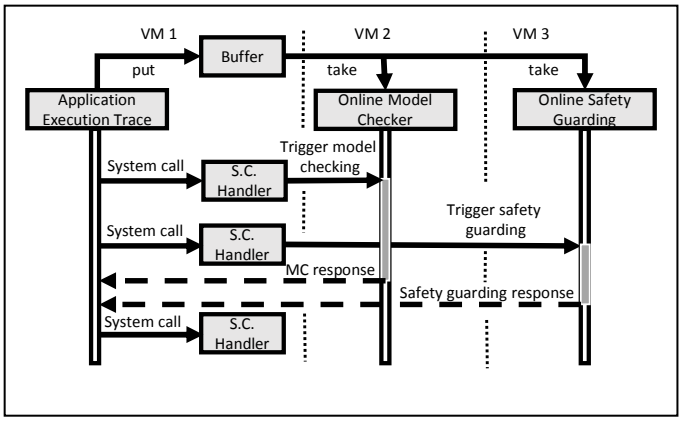

Fig. 3. Online Model Checking and Safety Guarding as OS Services via System Calls

IV. Self-adapting Safety Guards

A. Online Model Checking

In the context of this paper, Online Model Checking is used primarily as means to identify the need for online adaptation of application programs. In addition, it is perfectly applicable for checking the correctness of the resulting software after an adaptation. As this approach is covered in previous papers, e.g. [8], it will not be further discussed here.

B. Artificial Immune Systems

To realize self-adapting safety guards, the operating system requires mechanisms to cope with dynamic behavior and to monitor system behavior at run-time. Safety guards are then responsible for identifying of potentially malicious system states caused by autonomous adaptation. Our safety guards depend on a knowledge base of normal system behavior, which has to be generated at run-time without any predefined design-time system model. Incorporated directly into the operating system, again via the system call interface (see Fig. 3), the safety guards continuously update this knowledge base of system behavior.

We use the principle of the Artificial Immune System's Danger Theory to implement these safety guards into the operating system. So-called Dendritic Cells build up the core of this population-based approach used to monitor and evaluate the system behavior. One DC is responsible for profiling the behavior of one task or a specific OS kernel property by the sequence of its system calls. It concurrently a) builds up a local knowledge base of normal behavior of the component it is monitoring and b) performs local behavior evaluation, using a Suffix Tree-based pattern-matching algorithm. In accordance to Danger Theory, on top of this pattern matching, system-wide input signals support the DCs' local evaluation. These signals are provided by an RTOS health monitor. The following input signals are defined:

(1) Safe signal: indicates that no threat has been identified in the system.

(2) PAMP ("pathogen associated molecular pattern") signal: indicator that a known threat has been localized.

(3) Danger signal: indicates a potential danger due to local behavior deviations.

(4) Inflammation signal: general alarm signal.

The evaluation of a DC is combining the result of its local analyzing method with these system-wide signals.

Each DC continuously receives the values of these system-wide signals indicating either safe, suspicious or dangerous system state based on specific thresholds assigned to the input signals and thereby classifies the local behavior. The general alarm signal can be raised to inform the DCs about potential risk situations. Such a risk situation may be triggered either due to inconsistencies between the model and the reality, identified by online model checking, or because of adaptation.

V. VIRTUALIZATION

A. Concepts for Real-time-capable Virtualization

As an evolutionary approach, we propose to follow system virtualization, as for example realized by our real-time multicore hypervisor Proteus [17]. A hypervisor (also known as virtual machine monitor) allows the sharing of the underlying hardware among multiple isolated virtual machines (VM). Multiple existing software stacks of operating system (OS) and application tasks are combined to a system of systems. Proteus can act as a basic OS itself in order to host tasks directly in a bare-metal manner. Virtualization is a well-suited architecture for CPSs, primarily due to capabilities such as integration of legacy code, scalability, and transparent use of multi-core processors, cross-platform portability, and isolation of applications, especially for open systems, in which subsystems may be added or removed at runtime. System virtualization provides a natural way to support mixed criticality by consolidating systems of different criticality levels into separated VMs. The coexistence of mixed criticality levels (e.g. safety-critical, mission-critical and subsystems of minor importance) has been identified as one of the core foundational concepts for CPSs [18]. CPSs adjust their goals and behavior at runtime according to changes of the environment or corrections received from a higher level. This results in varying resource usage patterns and the need for a dynamic resource management. Proteus combines safe resource partitioning and dynamic reallocation of resources.

In previous work, the Flexible Resource Manager (FRM) has been developed in our group [19] and was recently adapted to system virtualization [20]. The FRM approach assumes that components are available in various profiles, which are functionally equivalent, but differ substantially regarding nonfunctional properties, especially resource requirements. Such implementation alternatives exist for example in case of optimization applications (relax optimality for lower resource consumption) or control applications (variable frequency). Profiles define minimum and maximum resource requirements for tasks and VMs: resource allocation is only possible within this range.

System virtualization implies resource management decisions on two levels and a FRM component is added to both the hypervisor and the OS. The hypervisor assigns resources to the VMs and the OSs assign the obtained resources to their tasks. The OS-FRMs inform the Hypervisor-FRM about the dynamic resource requirements and utilizations. The Hypervisor-FRM's resource allocation among the VMs is based on this information. The Hypervisor-FRM informs the OS-FRMs about the assigned resources, which facilitates each OS-FRM to manage its resource share. The dynamic switching by the FRMs between profiles on task level and on VM level implements the dynamic resource management. In particular, reserved but temporarily unused resources can be passed to other tasks/VMs. If resources were reallocated from one entity to another one and the lending entity later needs more resources than left, a resource conflict occurs and has to be solved in real-time. To achieve this, an acceptance test precedes each profile switch. Such a switch is accepted if and only if a feasible reconfiguration to a fallback configuration is identified, which fulfills the worst-case requirements of all entities, and if the time required performing this reconfiguration does not lead to a deadline miss.

So far, our work focused on the management of the resource computation time and showed that the actual distribution of bandwidth follows closely the desired bandwidths, even in case of varying execution times and mode changes, underlining the effectiveness of our approach in implementing an adaptive resource allocation [21].

B. Resulting implementation architecture

The above-mentioned virtualization concepts serve as a unified platform to implement self-adaptive systems. The basic idea is to run the primary hard real-time tasks grouped into one VM under highest criticality. This includes any access to primary (hard real-time) parameters and the monitoring of system calls needed to link online model checking and safety guarding. Access to secondary parameters, i.e. accessing global information via a network, can be assigned to a lower criticality virtual machine. As the tasks on this machine highly depend on the availability of messages from remote nodes, we make use of the dynamic bandwidth assignment feature of our virtualization concept; i.e. this virtual machine becomes scheduled with a certain fraction of the available processor bandwidth only when needed. The online model checker and the safety guarding run on an additional VM. Communication in between the respective application and these services is controlled by the cross-VM communication service of the hypervisor. An additional VM is reserved for strategic activities, including replacing/modification components when a necessity of adaptation has been identified. Again, by making use of the dynamic bandwidth assignment capabilities of our virtualization concept, whenever necessary we can assign as much bandwidth as possible to this VM after having set all other tasks running on all other VMs of the system to their least resource-hungry profile.

C. Results

The approach is currently integrated into our real-time multi-core hypervisor Proteus and our real-time operating system ORCOS for 32-bit multi-core PowerPC 405 architectures. A low memory footprint and a high configurability characterize Proteus. A configuration with the base functionality requires 11 kilobytes and a configuration with all functional features requires 15 kilobytes. The interrupt latencies and the execution times for synchronization primitives, hypercall handlers, emulation routines, and virtual machine context switch are all in the range of hundreds of processor cycles. Executed with a clock speed of 300 MHz, a virtual machine context switch takes 1.3 µs. Virtualization increases the interrupt latency. The additional latency is 0.5 µs for a programmable timer interrupt and 0.3 µs for a system call interrupt.

Proteus supports both, full virtualization and para-virtualization without relying on special hardware support.

The implementation of the FRM requires para-virtualization, since the OS-FRMs have to pass information to the hypervisor. The requirement to modify the guest OS is outweighed by the advantages in terms of efficiency, run-time flexibility, and cooperation of hypervisor. A specific advantage of para-virtualization for real-time systems is the possibility to apply dynamic real-time scheduling algorithms. Moreover, para-virtualization's replacement of privileged instructions by hypercalls speeds up the execution in average by 39%.

We tested the integration of online model checking in a rather abstracted emulation framework, just to enable some first rough analysis (see [22]). In this experiment, ORCOS with the application to be online checked runs on a QEMU-emulated Power PC405 on top of Ubuntu 12.4 LTS LINUX (which did replace the hypervisor in this experiment). The model checker became just an application running on Linux, while a special communication helper modeled the cross-VM communication service of the hypervisor. This experiment showed linear communication overhead with respect to the size of the transferred state information. The absolute value on a 3 GHz Intel P4 was in the low millisecond range, a value that will dramatically be lower in case of a real hypervisor-based implementation.

VI. Summary and Outlook

In this paper, we are addressing some system-engineering aspects of upcoming highly adaptable and self-modifying embedded real-time software. We are concentrating on platform aspects and two activities needed for self-adaptation: how Online Model Checking may be applied as a means to identify necessary module replacements/additions and how Artificial Immune Systems can be used for online safety guarding. Both activities are carried out at the granularity of system calls.

There are numerous open questions to be answered until such visionary self-adapting real-time systems can become reality, e.g.: (1) How to organize a market for components that can be handled in a fully automatic manner? (2) How to organize a repository of available components such that it can be queried automatically? (3) How to fully automatically convert a model checker's counterexample into a query for a problem-solving component? (4) How to implement a self-adaptation environment such that it can run under real-time constraints? Currently we are working on some of these questions.

Acknowledgment

Part of the work described in this paper has been carried out within CRC 614 "Self-Optimization in Mechanical Engineering" funded by the German Science Foundation, and project AIS (Autonomous Integrated Systems), funded by edacentrum, Hannover, Germany.

References

[1] M. Broy, A. Schmidt, "Challenges in Engineering Cyber-Physical Systems", IEEE Computer Vol. 47 no. 2, Feb. 2014, pp. 70-72

[2] J.O. Kephart, D.M. Chess, "The vision of autonomic computing", IEEE Computer 36(1), pp. 41-50, 2003

[3] Y. Brun, et al., "Engineering self-adaptive systems through feedback loops," in Proc. of Software Engineering for Self-Adaptive Systems, ser. LNCS 5525, B. H. C. Cheng and et al., Eds. Springer, 2009, pp. 48–70.

[4] F. Rammig, L. Khaluf, N. Montealegre, K. Stahl, Y. Zhao, "Organic Real-time Programming – Vision and Approaches towards Self-Evolving and Adaptive Real-time Software", in Proc. 9th IEEE SEUS 2013, , 17. - 18. Jun. 2013 IEEE

[5] Z. Huma, C. Gerth, G. Engels, and O. Juwig, "Towards an automatic service discovery for uml-based rich service descriptions," in Proc. ACM/IEEE 15th International Conference on Model Driven Engineering Languages and Systems (MODELS'12), ser. LNCS. Springer, 2012.

[6] R. Calinescu, C. Ghezzi, M. Kwiatkowska, and R. Mirandola, "Self-adaptive software needs quantitative verification at runtime," Commun. ACM, vol. 55, no. 9, pp. 69–77, Sep. 2012.

[7] F. J. Rammig, Y. Zhao, and S. Samara, "On-line model checking as operating system service," in Proceedings of the 7th IFIP WG 10.2 International Workshop on Software Technologies for Embedded and Ubiquitous Systems, ser. SEUS'09. Springer-Verlag, 2009, pp. 131–143.

[8] Y. Zhao and F.-J. Rammig, "Online model checking for dependable real-time systems," in 16th IEEE ISORC, China, IEEE Computer Society. IEEE Computer Society, April 2012, pp. 154–161.

[9] F. Rammig and K. Stahl, "Online Behavior Classification for Anomaly Detection in Self-x Real-Time Systems". in Proc. 5th IEEE SORT, 2014, , 9. Jun. 2014 IEEE.

[10] U. Aickelin and S. Cayzer. "The danger theory and its application to artificial immune systems". CoRR, 2008.

[11] L. Gausemeier, F.J. Rammig, W. Schäfer (Eds), "Design Methodology for Intelligent Technical Systems," Springer-Verlag, Heidelberg, Germany, Jan. 2014.

[12] Orna Kupferman and Moshe Y. Vardi, "Model checking of safety properties," Form. Methods Syst. Des., 19(3):291–314, October 2001. ISSN 0925-9856

[13] Viktor Schuppan, "Liveness checking as safety checking to find shortest counterexamples to linear time properties," PhD thesis, ETH Zurich, 2006.

[14] A. Biere, A. Cimatti, E. M. Clarke, O. Strichman, and Y. Zhu, "Bounded model checking," Advances in Computers, vol. 58, pp. 118–149, 2003.

[15] M. Qanadilo, S. Samara, and Y. Zhao, "Accelerating online model checking," 6th Latin-Am. Symp. on Dependable Comp. (LADC), 2013

[16] H. Kopetz, Real-time systems: design principles for distributed embedded applications, ser. Kluwer interntl. series in engineering and computer science: Real-time systems. Kluwer Academic Publ., 2011.

[17] D. Baldin, T. Kerstan, "Proteus, a hybrid Virtualization Platform for Embedded Systems", in: Analysis, Architectures and Modelling of Embedded Systems, 14. - 16. Sep. 2009 IFIP WG 10.5, Springer-Verlag

[18] S. Baruah, H. Li, and L. Stougie. "Towards the design of certifiable mixed-criticality systems",in Proc. IEEE Real-Time Technology and Applications Symposium, 2010

[19] S. Oberthür, L. Zaremba, and H. S. Lichte, "Flexible resource management for self-x systems: An evaluation," in Proceedings of the 2010 13th IEEE International Symposium on Object, Component, Service-Oriented Real-Time Distributed Computing Workshops, ser. ISORCW'10. Washington, DC, USA: IEEE CS, 2010, pp. 1–10.

[20] S. Groesbrink, S. Oberthür, D. Baldin, "Towards Adaptive Resource Management for Virtualized Real-Time Systems", in: 4th WS on Adaptive and Reconfigurable Embedded Systems (CPSWeek 2012),

[21] S. Groesbrink, L. Almeida, M. de Sousa, S.M. Petters, "Towards Certifiable Adaptive Reservations for Hypervisor-based Virtualization," in Proceedings of the 20th IEEE RTAS, April 2014

[22] K. Sudhakar,Y. Zhao, F.J. Rammig, "Efficient Integration of Online Model Checking into a Small-Footprint Real-time Operating System". in Proc. 5th IEEE SORT 2014, , 9. Jun. 2014 IEEE.