A Software Update Service with Self-Protection Capabilities

Moritz Neukirchner, Steffen Stein, Harald Schrom, Rolf Ernst
Institut für Datentechnik und Kommunikationsnetze
Technische Universität Braunschweig
Email: neukirchner|stein|schrom|ernst@ida.ing.tu-bs.de

***Abstract*—Integration of system components is a crucial challenge in the design of embedded real-time systems, as complex non-functional interdependencies may exist. We propose a software update service with self-protection capabilities against unverified system updates - thus solving the integration problem *in-system*.**

As modern embedded systems may evolve through software updates, component replacement or even self-optimization, possible system configurations are hard to predict. Thus the designer of system updates does not know the exact system configuration. This turns the proof of system feasibility into a critical challenge. This paper presents the architecture of a framework and associated protocols enabling updates in embedded systems while ensuring safe operation w.r.t. non-functional properties. The proposed process employs contract based principles at the interfaces towards applications to perform an in-system verification. Practical feasibility of our approach is demonstrated by an implementation of the update process, which is analzed w.r.t. the memory consumption overhead and execution time.

I. Introduction

One of the largest challenges in the design of larger embedded systems, such as an automotive or avionic platform, is the large number of variants and the continuous development by software updates in the field. Already today, modern upper-class automobiles can be ordered in up to 2^{25} configurations with possible effects on the electric system [1]. This is only possible because of software flexibility, but it turns integration into a hard task.

To cope with this problem, OEMs currently maintain a complex versioning database and perform exhaustive testing to cover the whole configuration landscape, and this has to be repeated for any update. This is a very time consuming and costly procedure and puts a high burden on the integrator (e.g. the car manufacturer) and its continuous collaboration with all suppliers. This strongly limits the possible evolution of an automotive embedded system over its lifetime.

An alternative is to move part of the design process to the field by establishing an in-field test bed and/or by including "self-x" (self-test, self-protection, self-optimization) functions in the systems that are based on formal models of the required system behaviour. In this case only the actual system configuration has to be verified. In this paper, we propose a modular software service for the in-field software update integration problem. We focus on performance guarantees because, due to resource sharing, integration leads to non-functional performance dependencies that are difficult to test, even in the lab. Therefore performance guarantees are key in integration.

The remainder of this paper is organized as follows. First, we will give an overview of related work and outline the methodology of managing contracts at runtime. Section IV will give a brief definition of terms and explain the principles of in-system contracting. Sections V and VI will elaborate on the architecture of our runtime environment (RTE) and on the associated protocol, respectively. Section VII will cover some aspects of the implementation and give an overview of the test environments used. In section VIII we will evaluate the performance of the developed framework, to then conclude the paper.

II. Related Work

The common approach to face the challenge of system integration is specification of each system component and verification of the complete system using the data from the specification. In recent years the component-based design methodology was established to provide means to design components and subsystems in isolation [2]. In *Contract Theory* a similiar approach is taken - system component descriptions are supplemented by assume/guarantee interfaces describing functional and non-functional properties and constraints [3]. The composition of these components can then be tested for compatibility and feasibility. Real-time extensions have been proposed, that also enable verification of compliance with real-time constraints using formal methods [4]. [5] proposed to use contracts to specify software components w.r.t. component behavior, interaction, synchronization and QoS attributes to verify component composibility.

We propose to use the contracting approach at runtime in the system itself. Our focus lies on hard real-time systems. Existing runtime implementations supporting contracting between system components w.r.t. performance metrics as described in [6] use periodic or total bandwidth servers [7] to ensure isolation between applications and adherence to contracts - thus employing orthogonalization, which tends to impose restrictions on resource utilization. In the context of embedded systems, [8] developed a contract-based framework to support self-optimization w.r.t. timing properties based on control theory. We are however not aware of any approach that offers performance contracting mechanisms on a system-wide

978-3-9810801-6-2/DATE10 © 2010 EDAA

level based on formal methods - thus allowing verification before execution and high utilization.

To allow contracting in the system, an additional software layer has to be introduced to embedded software to control and supervise access to platform resources. This trend towards multiple software layers in embedded systems could already be observed in recent years. The automotive industry has developed a standardized middleware API as one result of the AUTOSAR [9] initiative. Also, several research groups are working towards layered software architectures to manage applications in embedded systems on different levels (e.g. reliability, soft real-time, trust levels) [10], [8], [11].

The software service presented in this paper follows this trend of multiple software layers in embedded systems and aims to incorporate the established principle of contracts into such a system. The resulting RTE allows for self-protection capabilities against infeasible system changes.

III. Methodology

Because in-field test is strongly limited (e.g. no test drives), the approach presented here resorts to a formal update protocol, which assumes that the parties providing software updates are generally well behaving and that in case of a failure, it must be possible to determine liability.

We first assume that the individual software function to be updated has been thoroughly tested or analyzed in the lab and has been characterized with its execution time on the target hardware, required bandwidth, and memory usage. This is a realistic approach that is used for lab based integration today [12], [13]. We assume that the supplier is responsible for the correctness of the individual characterization. In the following integration process, these data are used as performance specification and are taken for granted, but are monitored to switch, e.g., to a fail-safe state if the specification turns out to be incorrect. In such a system, software function development, integration, and safety can be treated as separate concerns treated with separate mechanisms.

Contracts are used as interface between these mechanisms. Monitoring is used to watch adherence to contracts. If well defined, the contracts and monitoring can be used to determine liability in case of failure.

We propose to establish an RTE as abstraction layer between platform and application capable of managing such contracts. As we are concerned with performance metrics, contracts will refer to data relevant for system timing properties. We aim at preverifying the contracts using formal methods before admitting corresponding configurations to take effect in the system. Thus, the system will only transition between provenly safe configurations. We focus on the overall structure and the update protocol while individual modules, such as the online performance verification may be exchanged. The system model and the distributed performance analysis [14] are based on the industrial offline tool SymTA/S [15] and have previously been published. Architecture support, such as virtualization or temporal separation using. e.g., time boxing, are orthogonal methods and can support the self-protection protocol. This is beyond the scope of this paper.

IV. Contracting

In this section, we will elaborate on the principles of in-system contracting. To introduce the main terms and concepts used in this paper we will first provide a brief definition of terms.

The term *platform* refers to the hardware any software will run on. We assume that a platform consists of multiple interconnected processors. These processors will be connected by communication media. We will refer to processors and buses as (computational and communication) *resources*.

We assume that a runtime environment is running on each processor of this platform. We propose that the framework described in this paper resides within this runtime environment. Furthermore, we expect the RTE to contain an operating system capable of multi-tasking.

On top of the runtime environment, *applications* are running. An application consists of a set of potentially communicating *tasks*. By task we understand a piece of code that will be scheduled by the operating system within the RTE. Note that although an application may span multiple processors it does not necessarily use all processors available in the platform.

In addition to the term application, we will use the term *system* to describe the combination of a platform running an RTE on each processor as described above, that may run a set of applications. We will also use the terms application and system in order to describe the availability of information within the RTE. If information is considered to be available (or synchronized) *application wide*, we assume that the information is available at each RTE instance that hosts at least one task of the given application. Similarly, information is available *system wide* if all RTE instances have access to it.

We refer to the binary data that contains the executable code of a task as the *task code*. Note that the code of multiple tasks may be provided as a single file.

In order to be able to configure the RTE and verify system properties at runtime, we assume that additional meta information is provided along with an application. We refer to this description as *annotations*.

In line with work in interface based design [16], by a *contract*, we understand a set of prerequisites an application requires and a set of guarantees it can give in turn. In our context of performance verification, the guarantees of application contracts consist of a performance characterization of the application in isolation, as specified in the annotations. The prerequisites are a set of constraints the application assumes to adhere to. The contracts include:

- A mapping of the tasks to the computational resources
- The task graph describing the inter-task communication
- Timing and activation models for tasks and communication links (bcet, wcet, period, jitter, minimum distance)
- Constraints on end-to-end latencies between tasks

If a contract is accepted, the RTE guarantees that all applications comply to their constraints given they do behave as

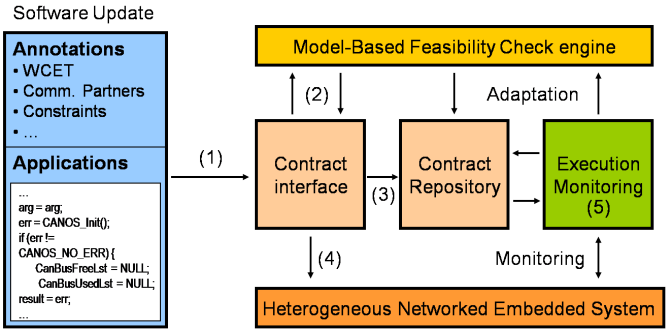


Fig. 1: Contracting Flow within RTE

described in the contract. The RTE becomes a contract broker, managing contracts with applications in order to guarantee undisturbed operation of the system.

In order to negotiate and verify contracts in the system itself, we established a flow as depicted in figure 1. The key components are a *Contract Interface* and a *Contract Repository*. An application and its associated contracts are presented to the system using the contract interface (1). The software component responsible for negotiating the contracts then inserts the contract data into the current system model within the feasibility evaluator in order to determine feasibility (2). In case an application can be accepted, its contract is stored in the Contract Repository (3) and it is admitted to execute on the system (4). Once the application is running, execution monitoring components (5) supervise adherence to the service contracts and can notify a controlling software component in case a service contract is broken. Runtime monitoring of timing properties, as used here, has been studied intensively in the past [17], [18], [19] and is not further discussed in this paper.

We assume that a new or updated application is loaded into a central entity of the system from an external source. This entity, the *Update Controller*, will distribute the annotations of the application to the appropriate resources. Every instance of the distributed feasibility evaluator (i.e. the analysis engine) only receives data of tasks or task links that it has to analyze. In case the application is accepted the Update Controller will distribute the task codes to the appropriate resources. Note that although an update controller is a central instance for the time needed for any given update process, it may be executed on any computational resource of the platform, potentially on a different one for each update process.

V. Framework Architecture

In the scope of the EPOC project, we are implementing a system as outlined above. The focus lies on developing a suitable software architecture for contract validation and enforcement. This backing framework is able to handle annotations, transform and insert them into an analysable model for a model-based performance analysis engine and deduce a decision whether the application can be accepted into the system or not.

As we target in-system verification of real-time systems, reliance on formally deduced application models becomes inevitable. Model analysis and contract negotiation should have minimal impact on execution of accepted applications though. Thus, the framework architecture, as shown in figure 2, is strictly separated into two domains, the *Model Domain* (left) and the *Execution Domain* (right). While the former solely operates on application models the latter enforces the parameter settings according to the model. To ensure consistency of the model and the execution parameters of accepted applications the Contract Repository acts as an interface between the two domains.

Furthermore, the strict decoupling of model and execution domain has the advantage that system configurations can be evaluated in the model domain without the necessity of loading the actual application onto the embedded system. This approach limits the temporary impact on required resources, such as memory, on the system to a minimum, in case an application or update is rejected.

The core component of the Model Domain is the Model Analysis. It performs the feasibility check. In our case it is based on a distributed performance analysis algorithm [14], while also an external centralized analysis component could be used. Models, as required by the analysis, are supplied by the Contract Negotiation component. It accepts application specifications and requirements via the Contract Interface, transforms them into an analysable format and interprets the results of the verification process. Closed contracts are stored in the Contract Repository. This feasibility check can be extended by a model-based optimizer to close a control-loop in the model domain, enabling runtime optimization.

The Execution Domain, as depicted on the right hand side of figure 2, enforces application contracts by setting execution parameters of applications and supervising task behaviour. To perform this task the Contract Enforcement reads accepted contracts from the Contract Repository and configures the platform accordingly (e.g. setting task and communication priorities). Metrics about the actual behaviour of running software components are collected by Contract Supervision and compared to the expected behaviour. If violations are detected they are reported back to Contract Enforcement which takes appropriate measures.

Both domains in combination ensure safe evolution of the system, as new configurations can be verified and evaluated w.r.t. to their performance while running applications can be monitored and controlled according to their model.

VI. Update Procedure

In order to manage dynamic software updates in an evolving system, we developed a lightweight protocol which in the following will be referred to as *Update Protocol*. The Update Protocol is initiated by the *Update Controller*. Every time it receives a new application model from an external source, it distributes the annotations to all affected RTE instances. The resulting configuration is analyzed distributedly and can be optimized if necessary. The *Contract Negotiation* sends a

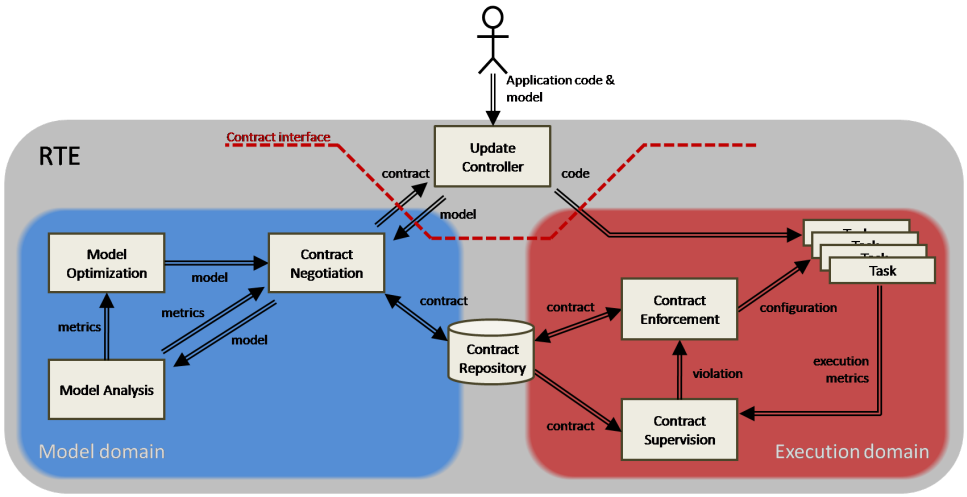


Fig. 2: Framework Architecture

resulting performance guarantee back to the Update Controller if all constraints of all applications are satisfied. At this point the feasibility of the new system configuration is determined. If the contract is closed, it is stored in the Contract Repository and the Update Controller transfers the application code to the corresponding processors. In the following step *Contract Enforcement* starts execution of the new application with parameters obtained from the Contract Repository.

Note that only one Update Controller is active during the update procedure, whereas the Model Domain as well as the Execution Domain components are active on all resources, forming a distributed evaluation and control system of cooperating software agents.

As Model and Execution Domain are composed of cooperating software agents that are each unaware of the global system state, consistency poses an issue on several levels. First, prior to system analysis, consistency of application models across the affected resources has to be assured. In a second step system analysis results have to be synchronized. Finally all affected resources have to switch to the new system configuration in a synchronized manner to prevent execution of application components of different versions.

A state synchronization mechanism which performs a barrier synchronization along the application's task graph ensures this consistency. It is implemented completely distributed and is applicable to evolving applications to match the overall framework concept.

We will now focus on the specific protocol that is executed during the different phases of the update procedure (figure 3). It consists of three main sections separated by synchronisation procedures as described above to ensure data consistency across all computational resources. In the first part the application model is transferred to all affected computational resources. In the second part the analysis is performed and, in case the new configuration is proven feasible, the application code is transferred. During the last part of the sequence all settings and execution parameters are enforced.

In the first step the Update Controller transfers the appli-

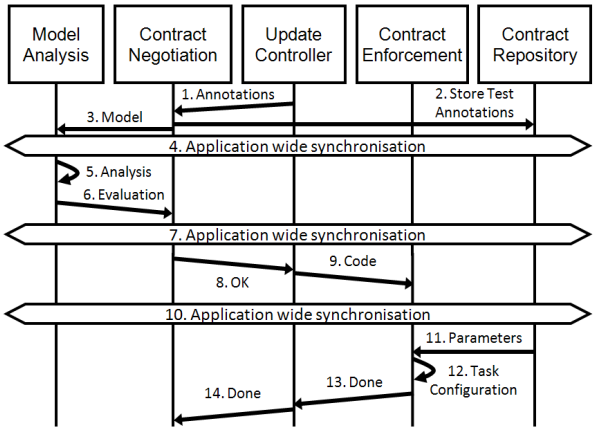


Fig. 3: Message sequence chart of update process

cation model to the *Contract Negotiation Components* of all RTE instances affected by the update. The transfer is realized using the *Contract Interface* (1). Each Contract Negotiation Component stores only a partial model in its local *Contract Repository* (2). Furthermore, it inserts the model in the *Model Analysis Component* (3) which later performs the system verification. The *Contract Repository* serves as a database for all contracts. It stores accepted contracts, as well as contracts that are yet to be evaluated. An *application wide synchronisation* (4) ensures model consistency across the distributed system.

At the beginning of the second phase all necessary data is available application wide, therefore the *model-based analysis* (5) can be performed. If the analysis is implemented distributedly, as in our implementation, this step requires cooperation of the analysis component instances. This process is not shown explicitly in figure 3. After the analysis process has finished, the results are reported back (6) to the Contract Negotiation Components that check the feasibility. In the following synchronization step consistency of the analysis results is ensured application-wide (7). The results are then forwarded (8) to the Update Controller. If all analysis results are positive, the

update is accepted and the Update Controller transfers the *program code* (9) to the appropriate computational resources. A following *application wide synchronisation* (10) assures the completion of the code transfer.

In the last part of the update procedure the Contract Enforcement reads the new *execution parameters* (11) from the Contract Repository to configure the communication resources and set the task execution parameters (11). Furthermore the task execution is started by setting the appropriate states in the task manager. As soon as all tasks are running a *notification* (13) is send by all Contract Enforcement Components to the Update Controller to signal completion. When all notifications are obtained, the Update Controller notifies all Contract Negotiation Components that the update process is completed (14). Only after this final notification new updates can be accepted.

VII. Implementation & Test

We have implemented all components used in the Update Procedure (figure 3) as elaborated above. The Model Analysis component is an implementation of the distributed performance analysis as described in [14]. uC/OS-II [20] is used as micro kernel. The code is written in plain C.

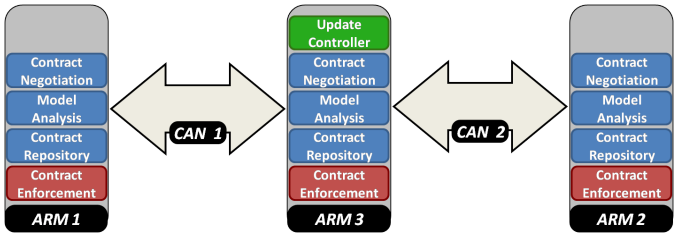
In order to give an idea of the overhead of such a framework we have analyzed the memory consumption. The first implementation of the framework uses about 270kB of memory for program code of which 32kB are used by the micro kernel. Additionally 105kB of memory is allocated for data of which most is part of the Contract Repository. The data size of the Contract Repository however can easily be scaled via defines depending on the number of tasks a resource has to be able to accommodate.

Two different test platforms are used to evaluate functionality and performance of the proposed framework.

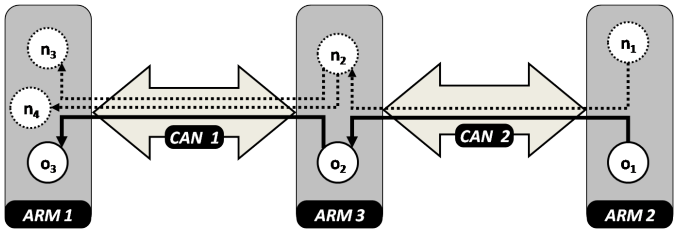
For investigation of functionality in a real-world setup a hardware platform consisting of two Microsys PM520 evaluation boards, that are connected via a CAN-Bus, is used. Both boards are equipped with a Freescale MPC5200 microcontroller.

The second test platform used, is the cycle-accurate hardware simulator CoMET by VaST Systems Technology [21]. The setup allows for simulation of larger hardware setups and at the same time provides means for measurement of execution times on all processors. All experiments presented in this work were performed on this platform. The simulated hardware environment consists of three ARM926E processors running at 200MHz which are connected via two CAN busses (fig. 4). Figure 4a shows the setup of the framework components of the test system. Contract Negotiation, Model Analysis, Contract Repository and Contract Enforcement exist on all resources and cooperatively perform their respective tasks. The Update Manager resides solely on ARM 3.

The setup of the applications is depicted in figure 4b. In the initial state all user tasks (o_1-o_3, n_1-n_4) and their models are compiled into the code of *ARM 3*. The user tasks are not running and all RTE instances are unaware of the corresponding model information. An excerpt of the model



(a) Framework structure of test system



(b) Application structure of test system

Fig. 4: Setup of test system

Task ID	*Prio.*	*Act. period*	*Act. jitter*	*WCET*	*Comm. partners*
o_1	2	500	5	10	o_2
o_2	2	n.a.	n.a.	10	o_1, o_3
o_3	3	n.a.	n.a.	17	o_2
n_1	1	500	0	10	n_2
n_2	1	n.a.	n.a.	15	n_1, n_3, n_4
n_3	1	n.a.	n.a.	17	n_2
n_4	2	n.a.	n.a.	13	n_2

Link ID	*Prio.*	*Act. period*	*Act. jitter*	*WCET*	*src. task*	*trg. task*
o_1-o_2	2	n.a.	n.a.	120	o_1	o_2
o_2-o_3	3	n.a.	n.a.	120	o_2	o_3
n_1-n_2	1	n.a.	n.a.	150	n_1	n_2
n_2-n_3	1	n.a.	n.a.	50	n_2	n_3
n_2-n_4	2	n.a.	n.a.	150	n_2	n_4.

TABLE I: Model specification and performance requirements (Act. period = Activation period [ms], Act. jitter = Activation jitter [ms], WCET = worst-case execution time [ms], src. = source, trg. = target)

specification of all tasks and communication channels of both applications is given in table I. The required end-to-end latency between o_1 and o_3 is 300 ms, between n_1 and n_3 500 ms and between n_1 and n_4 also 500 ms. In a first step the application consisting of tasks o_1, o_2 and o_3 is inserted into the framework following the update process as depicted in fig. 3. As no other application is present in the system the latency along the path from o_1 to o_3 simply equals to the sum of the WCETs of its tasks and communication channels. The resulting latency of 277 ms is lower than the specified constraint of 300 ms. Thus the first application is admitted to the system and starts execution.

After the first application is accepted the model of the

second application consisting of tasks n_1, n_2, n_3 and n_4 is added to the system model (fig. 4). As the second application uses the highest priorities in the system all other tasks and communication channels would be influenced in their timing behavior. The analysis returns path latencies of 687 ms between o_1 and o_3, 247 ms between n_1 and n_3 and 410 ms between n_1 and n_4. As the latency constraint of the path between o_1 and o_3 is violated the second application is rejected and thus its update process is aborted after step 7 of fig. 3.

VIII. Evaluation

In this section we will present the information on the execution time of the update process for the given test system. Timing information has been acquired with CoMET. All references to steps in the update process refer to the corresponding labels in fig. 3.

The entire process (steps 1-14) for inserting the first application required approx. 426 ms. While negotiating the contract (steps 1-8) took 143 ms, the remaining 283 ms were used for distributing and starting the application (steps 9-14).

Further analysis of the execution time distribution showed that large portions of these time windows were caused through the fairly slow CAN bus.

The 143 ms for contract negotiation were composed of 108 ms for transmission of the model via the CAN bus (step 1), 23 ms for synchronization (steps 4 and 7) and 9 ms for the distributed analysis itself (step 5). Synchronization and analysis both use the CAN bus for communication as well. The remaining 3 ms were used for protocol handling and local model operations.

The execution domain operations of the update process (steps 9-14) consist primarily of transmission of the program code over the CAN bus (step 9), which required 259 ms, and a synchronization step (step 10), that took 23 ms to execute. During the remaining 1 ms task parameters were set, communication channels established and protocol handling was performed.

The values for the contract negotiation of the second application are comparable. Contract negotiation (steps 1-8) required 219 ms due to the larger model of the second application. Analysis was only slightly slower (12 ms) which is also due to the larger model, which led to an increased number of CAN messages. As the second application is rejected because of violated timing constraints the update process was aborted after the negotiation process.

As can be seen from the above execution time analysis, the proposed framework imposes an overhead of only some 100 ms, while these times are primarily influenced by the transmission speed of the used communication medium. Thus it is highly suitable for the targeted scope of application - in-field verification of system updates.

IX. Conclusion

In this paper we have presented a software update service with self-protection capabilities. Using contracts between applications and the runtime environment, component integration can be verified autonomously by the system itself. The presented contracting API provides the user with a uniform interface for specifying behaviour of applications and corresponding performance constraints.

As our architecture and the associated update protocol strictly separate application models and execution, the underlying verification process is decoupled to a large degree from running applications. Thus, new system configurations can be verified prior to their execution, maintaining a safe system state at all times.

The prototype implementation shows that such a framework can be implemented with low overhead concerning memory footprint and processor allocation. Thus the presented architecture is suitable for the targeted domain of in-field verification of system updates in distributed embedded systems.

References

[1] V. Feil, C. Heinisch, and M. Simons, "Betrachtungen zur komplexität bei der aktualisierung von software im automobil," 2005.

[2] W. Cesario, A. Baghdadi, L. Gauthier, D. Lyonnard, G. Nicolescu, Y. Paviot, S. Yoo, A. Jerraya, and M. Diaz-Nava, "Component-based design approach for multicore socs," in *Design Automation Conference, 2002. Proceedings. 39th*, 2002.

[3] T. A. Henzinger and S. Matic, "An interface algebra for real-time components," in *Proceedings of the 12th Annual Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2006.

[4] L. Thiele, E. Wandeler, and N. Stoimenov, "Real-time interfaces for composing real-time systems," in *6th ACM & IEEE Conference on Embedded Software (EMSOFT)*, 2006.

[5] S. Li, J. Wu, and Z. Hu, "A contract-based component model for embedded systems," in *Proceedings of the 4th ACM international conference on Quality Software*, 2004.

[6] S. Wang, S. Rho, Z. Mai, R. Bettati, and W. Zhao, "Real-time component-based systems," in *Proc. of the 11th IEEE Realtime and Embedded Technology and Applications Symposium (RTAS)*, 2005.

[7] J. Liu, *Real-time Systems*. Prentice Hall, 2000.

[8] FRESCOR: Framework for Real-time Embedded Systems based on COntRacts, "http://www.frescor.org," Internet.

[9] AUTOSAR - Automotive Open System Architecture, "http://www.autosar.org/," Internet.

[10] C. Haubelt, D. Koch, and J. Teich, "Reconet: modeling and implementation of fault tolerant distributed reconfigurable hardware," in *16th Symposium on Integrated Circuits and Systems Design*, 2003.

[11] Trust4All, "http://www.hitech-projects.com/euprojects/trust4all/," Internet.

[12] H. Espinoza, K. Richer, and S. Gerard, "Evaluating MARTE in an Industry-Driven Environment: TIMMO's Challenges for AUTOSAR Timing Modeling," in *Conf. on Design, Automation and Test in Europe (DATE), MARTE Workshop*, 2008.

[13] A. Koudri, D. Aulagnier, D. Vojtisek, P. Soulard, C. Moy, J. Champeu, J. Vidal, and J.-C. L. Lann, "Using marte in a co-design methodology," in *Conf. on Design, Automation and Test in Europe (DATE), MARTE Workshop*, 2008.

[14] S. Stein, A. Hamann, and R. Ernst, "Real-time property verification in organic computing systems," in *2nd IEEE International Symposium on Leveraging Applications of Formal Methods, Verification and Validation (ISoLA)*, 2006.

[15] R. Henia, A. Hamann, M. Jersak, R. Racu, K. Richter, and R. Ernst, "System level performance analysis - the symta/s approach," in *IEE Proceedings Computers and Digital Techniques*, 2005.

[16] L. de Alfaro and T. A. Henzinger, "Interface-based design," *Engineering Theories of Software-Intensive Systems*, vol. 195, 2005.

[17] D. Haban and K. Shin, "Application of real-time monitoring to scheduling tasks with random execution times," 1990.

[18] F. Jahanian, R. Rajkumar, and S. C. V. Raju, "Runtime monitoring of timing constraints in distributed real-time systems," *Real-Time Syst.*, vol. 7, pp. 247–273, 1994.

[19] A. Mok and G. Liu, "Efficient run-time monitoring of timing constraints," Jun 1997, pp. 252–262.

[20] J. J. Labrosse, *MicroC/OS-II: The Real-Time Kernel*. Mcgraw-Hill, 2002.

[21] VaST Systems Technology, "http://www.vastsystems.com," Internet.