Real-time Property Verification in Organic Computing Systems

Steffen Stein, Arne Hamann, Rolf Ernst
Institute of Computer and Communication Network Engineering
Technical University of Braunschweig
D-38106 Braunschweig / Germany
{stein | hamann | ernst}@ida.ing.tu-bs.de

***Abstract*— Integrating new functionality into complex embedded hard real-time systems requires considerable engineering effort. Emerging formal analysis methodologies and tools from real-time research assist system engineers solving this integration problem. For future organic computer systems, however, it is desirable to integrate these approaches into running systems, enabling them to autonomously perform e.g. online acceptance tests and self-optimization in case of system or environmental changes. This results in high system robustness and extensibility without explicit engineering effort.**

In this paper, we present an approach adapting formal compositional analysis techniques to realize self-awareness and self-adaptation in embedded systems with respect to real-time properties such as latency constraints, buffer sizes, etc.

We introduce a framework for distributed online performance analysis running on embedded real-time systems. Based on this framework we implement an acceptance test for the integration of new functionality into an existing embedded real-time system. Furthermore, we present an online optimization algorithm based on the same framework. In a case study, we demonstrate the applicability of the approach and show that online optimization can increase the acceptance rate with reasonable computational effort.

I. Introduction

Organic Computing [10] has recently emerged as a new challenge in computer engineering. As ubiquitous and embedded computing systems become increasingly powerful, the development paradigms shift from implementing the technically possible to building robust and easily usable systems. Organic computing systems tackle this challenge by introducing self-adaption, learning and self-configuration into future computer systems. Initiatives as IBMs Autonomic Computing Initiative [8] or Intels Proactive Computing [15] show that introducing self-configuration in complex computer systems is not only an academic endeavour.

Approaches to increase system robustness include the migration of tasks in a system, once a processing unit fails [14]. The approach presented uses shadow tasks and checkpointing, so that the shadow tasks can take over functionality if a primary task stops to execute. This introduces a high overhead and robustness is guaranteed by redundancy. Organic computing systems however should not only be able to resort to redundancy, but to autonomously reconfigure themselves and adapt to new environmental conditions as could be imposed by i.e. subsystem failures.

This paper will focus on the real-time properties of organic computing systems. In many applications, simple process deadlines are insufficient. Rather, end-to-end deadlines, jitter, and transient load requirements are needed for high quality media or control engineering. Furthermore, a networked system will integrate many functions on the same system platform, many of them with emergent behavior. The organic computing system will have to control the system emergence induced by self-configuration and -adaptation in such a way that these complex timing constraints are met at all times. Here, we can identify three challenges. The system must be aware of the current architecture and the current set of applications on the organic computing system, it must anticipate the effect of a planned reconfiguration and, finally, there must be an online mechanism to control emergence.

Many online algorithms for real-time analysis have been proposed [1], [4] because they are often needed for dynamic scheduling strategies. However, these techniques have not found their way into industrial applications, since the underlying application models are quite simple and do not reflect the complex timing behaviour of realistic heterogeneous distributed embedded systems. Therefore, complex timing constraints for embedded systems are usually verified by offline simulation or prototyping. Those techniques, however, are not applicable in organic computing systems.

In the last couple of years, offline analysis algorithms [13], [3] have been proposed that can analyze complicated systems and constraints requiring access to the global system state. They have been used in system sensitivity analysis [12], [2] and system optimization [6]. In this paper, we propose a first step to extending such analysis to an online analysis and optimization system for organic computing systems.

The remainder of the paper is organized as follows. Section II introduces the basic concepts of offline performance analysis used as basis for our online analysis approach. Afterwards, in Section III, the underlying ideas of our distributed performance analysis are presented. In Sections IV and V we then introduce algorithms needed to perform an acceptance test and self-adaptation for integrating new functionalities into a system. In Section VI we demonstrate the applicability of the approach and show that self-adaptation can increase the acceptance rate.

II. Offline Compositional System-level Performance Analysis

In this section we give a brief overview over the compositional performance verification methodology proposed by

Richter et al. [13], [7], called SymTA/S, which we extend in this paper to realize a decentralized distributed online analysis and self-adaptation for complex heterogeneous embedded systems. We give a short summary of the utilized application model and outline the SymTA/S analysis methodology.

In SymTA/S, systems are modeled as (cyclic) task graphs which are mapped on computational and communication resources. A task is activated due to activating events. These can be generated in a multitude of ways, including expiration of a timer, external or internal interrupt, and task chaining. Each task is assumed to have one input FIFO. A task reads its activating data from its input FIFO and writes data into the input FIFO of a dependent task. A task may read its input data at any time during one execution. The data is therefore assumed to be available at the input during the whole execution of the task. SymTA/S also assumes that input data is removed from the input FIFO at the end of one execution.

A task needs to be mapped on a *computation* or *communication resource* to execute. If multiple tasks share the same resource, then two or more tasks may request the resource at the same time. In order to arbitrate request conflicts, a resource is associated with a *scheduler* which selects a task to which it grants the resource out of the set of active tasks according to some scheduling policy. *Scheduling analysis* calculates worst-case (sometimes also best-case) task response times, i.e. the time between task activation and task completion, for all tasks sharing a resource under the control of a scheduler. Scheduling analysis guarantees that all observable response times will fall into the calculated [best-case, worst-case] interval. Scheduling analysis is therefore conservative. A task is assumed to write its output data at the end of one execution. This assumption is standard in scheduling analysis.

SymTA/S solves the global system-level scheduling analysis problem by alternating (classical) local scheduling analyses and *event model propagation* along *event streams* connecting functionally dependant tasks.

Event models describe the possible I/O timing of tasks. *Input event models* capture event patterns leading to task activations. Based on *input event models* for all tasks mapped on a resource, local scheduling analysis can be performed and *output event models* can be derived. These *output event models* are propagated to connected resources, where they are used, in turn, as input event models for local scheduling analysis.

A popular event model is the so-called *standard event model*. For example, a *periodic* event model has one parameter $\mathcal{P}$ and states that each event exactly arrives periodically every $\mathcal{P}$ time units. This simple model can be extended with the notion of jitter, leading to a *periodic with jitter* event model. Such an event model is described by two parameters $(\mathcal{P}, \mathcal{J})$. It generally occurs periodically, but it can jitter around its exact position within a jitter interval $\mathcal{J}$. If the jitter is larger than the period, then two or more events can occur at the same time, leading to bursts. To describe a *bursty* event model, the *periodic with jitter* event model can be extended with a d^- parameter that captures the minimum distance between events within a burst.

More details about the compositional system-level performance analysis can be found in [13], [7].

Event models only capture key integration aspects such as send/receive message jitters, etc., and ignore details of the concrete computation or communication resources. They therefore represent a suitable abstraction for the realization of a light-weight distributed online system-level performance analysis, which will be discussed in the following sections.

III. Distributed Online Performance Analysis

In this section, we first introduce an example system that acts as a motivating example for our algorithms presented in the following sections. Afterwards, we will present our approach to distributedly model an applications running on an embedded system. Last, we introduce our strategy to analyse this distributed model.

A. Motivating Example

Consider the example setup shown in Figure 1. It contains four different computational units connected by a bus. Two applications are already mapped on the architecture. A video application (solid chain) gathers data from a camera controlled by the micro controller uC, performs preprocessing on the DSP and post-processing on the PPC core. The second application (dashed chain) reads data from a sensor, which is first processed on the ARM core and then forwarded to the PPC core, which, in turn, controls an actor. Both applications have constrained end-to-end latencies (Table I(c)).

Suppose the ARM processor as well as the DSP are not fully loaded with these applications, leaving room for integrating a third application into the system. We propose the integration of a second streaming application that runs on the ARM processor core and also uses the DSP for processing. In Figure 1, the task chain is shown under the block diagram, the white arrows indicate the desired mapping of the tasks on the system architecture. This application also has a constrained end-to-end latency (Table I(c)).

The computational resources (PPC, uC, DSP, and ARM) are all scheduled according to the static priority preemptive policy, and the interconnecting bus is arbitrated by the CAN protocol. Core execution and core communication times as well as priorities of all tasks and exchanged messages are given in Tables I(a) and I(b), respectively. The periods of incoming data at the system inputs (Cam, $Sens$, and S_{in}) are specified in Table I(d).

Channel	*CCT*	*Priority*
C0	[10.4, 12.4]	4
C1	[12, 14.4]	2
C2	[20, 26.4]	1
C3	[15.2, 22.4]	3
C4	[10.4, 14.4]	5
C5	[15.2, 26.4]	6

(a) Core Communication Times

Task	*CET*	*Priority*
T5	[30.9, 43.1]	1
T2	[15.8, 27.4]	2
T3	[36.9, 46.3]	1
T0	[20.1, 48.5]	2
T4	[13, 86]	1
T7	[40.3, 44.5]	2
T1	[27.1, 154.9]	1
T6	[14.2, 63.6]	2
T8	[11.5, 291.2]	3

(b) Core Execution Times

Path	*Deadline*
$Sens \rightarrow Act$	850
$Cam \rightarrow V_{out}$	1100
$S_{in} \rightarrow S_{out}$	1000

(c) End-to-End Constraints

System Input	*Period* $\mathcal{P}$
$Sens$	500
Cam	100
S_{in}	1000

(d) Input Event Models

TABLE I - System Parameters

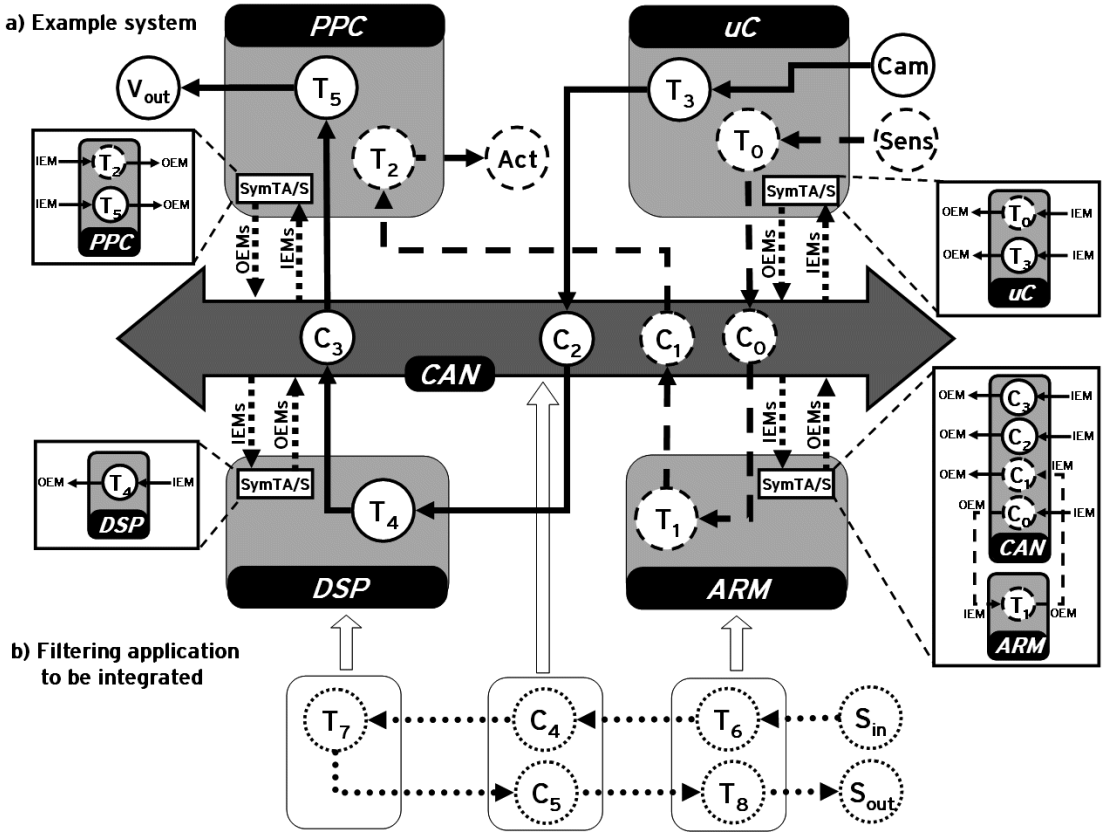

Fig. 1 - EXAMPLE SYSTEM ARCHITECTURE

The key challenge is to successfully integrate the new functionality into the system without violating the timing constraints of the already mapped applications. Offline performance analysis as introduced in the previous section offers techniques to verify timing behavior prior to integration. Currently, these methodologies are only integrated in offline analysis tools (e.g. SymTA/S [5]), thus new functionality can only be safely integrated after modeling, testing, and possibly adaptation of the remaining system by an engineer.

In order to enable integration of new functionality by non-experts, we suggest to integrate performance analysis methodologies on the embedded system itself. Our approach to online distributed performance analysis suggests, that on every computational resource in the system, a performance analysis application instance is running. Each of these instances has knowledge of part of the global model corresponding to the application running on the system.

In Figure 1 these analysis instances are depicted using white boxes labeled "SymTA/S". The boxes next to them show the parts of the global model to be analyzed that are known to the respective instances. Notice that each instance has a model of the tasks running on the same processing unit as itself. As the bus has no computational power that can be used for analysis, one instance has to take over its analysis - in this case the instance on the ARM core.

A distributed online performance analysis using the instances running on the embedded system, yields runtime performance data that can be used to implement an online acceptance test for the integration of new functionality into an embedded system. The same data can also be used to implement distributed optimization algorithms automatically adapting system parameters to changing system requirements and extended functionalities.

In the following sections we describe our approach to distributed online performance analysis and outline an acceptance test algorithm (Section IV) as well as an online optimization algorithm (Section V). These algorithms will be used in Section VI to successfully integrate the filtering application presented in Figure 1(b) into the system described in Figure 1(a).

B. Distributed system modeling

For distributed online analysis, distributed models of the application running on the embedded system as well as its architecture are needed for the analysis instances running on the computational resources. Basically, it is desirable that the analysis instances themselves are capable of generating these models using data obtainable from the system or additional information associated with the tasks running on the system such as best and worst case execution times and communication partners. This way, the designer effort for changing the system specification or integrating new functionalities into a running system would be minimized.

However, an automatic generation of these models from measured data in an embedded system is a complex problem and not in the scope of this paper. Therefore, we assume in the remainder of this paper, that there exists a complete distributed model of the architecture of the embedded system

and the application running on it. Additionally, we assume that new functionalities, which are integrated into the system come along with an appropriate model.

The basic compositional performance verification model, as used in the SymTA/S tool suite [13], [7], [5], needs to be extended to support distributed descriptions of a real-time system. We introduce the concept of *Remote Resources* to model remote parts of the model that are connected to the local instance. Remote Resources are containers that represent remote parts of the model specified in other instances of the analysis framework.

Since two partial models may be connected by multiple event streams, each Remote Resource may contain multiple *External Sinks* and *External Sources*. An External Sink-Source pair acts as a tunnel for analysis information (i.e. event models) being passed along the connected event streams. Note that the actual exchange of event model data is achieved over communication resources of the underlying architecture. In the case of the example system in Figure 1 the CAN-bus is used to exchange the necessary input event models (IEMs) and output event models (OEMs).

C. Analysis strategy

As described in Section II, our proposed analysis methodology alternates local scheduling analysis on resources and event model propagation along event streams. The order in which resources are analyzed can be chosen arbitrarily. During offline analysis the order is determined based on topological information to minimize computational effort.

For a distributed system, we suggest to analyze each resource as soon as new input event model data is available for at least one task mapped on the resource. This approach does not require any topological knowledge or central coordination of the analysis. In the remainder of the paper we refer to this approach as "live analysis". Not taking into account topological information for determining analysis order leads to an increase in required local analysis runs. This will be discussed in more detail in section VI.

A problem that arises from this "live" distributed setup is to determine analysis convergence. As new functionality can be added at random points of time, successive analysis iterations may be caused by an ongoing analysis or the integration of new functionality at a remote part of the system. Thus, it is hard to determine when analysis has converged, and supplies valid performance data for the current system setup.

However, in existing systems an analysis run converges significantly faster compared to the average time interval between system changes. For instance, the software of a car may be updated every few months, whereas an analysis run converges in seconds. Consequently, it is safe to assume that analysis has converged if no resource in the local instance had to be analyzed within a reasonable timeout interval.

Concurrently analysing a distributed system using the resources of the system itself introduces some computational and communication overhead into the system. The load imposed by live analysis will however be transient as it is only triggered by system model modifications. Furthermore, we aim to minimize interference by using idle times of the processors and buses for online analysis.

IV. Acceptance Test for new Functionality

Acceptance tests are a vital tool for real-time system integration. A new function can be accepted to be integrated into a real-time system if its integration does not overload the system, does not cause constraint violations for applications already running on the system, and meets its own constraints.

Implementing an acceptance test on embedded devices themselves adds self-protection properties to these systems, as they can deny modifications compromising their current functionality.

For the purpose of implementing an acceptance test using the performance analysis methodologies described above, we need to be able to specify constraints for distributed applications that affect multiple analysis instances. Similarly, it must be possible to determine global end-to-end latencies for these distributed applications.

In this section, we introduce techniques to distribute constraints throughout the system and collect global information about, for instance latencies or aggregated buffer sizes along paths. For simplicity, we outline the algorithms for constrained path latencies. Note that the same algorithms are also applicable for other performance data.

For distributing constraints inside the system, we assume that there exists a broadcast mechanism to send messages to all analysis instances currently present in the system. Techniques to establish communication paths and broadcasting mechanisms in unknown multihop communication topologies have been discussed in great detail in the field of wireless ad hoc networks (i.e. [11], [9]). These approaches can be adapted to establish the communication mechanisms claimed above.

Constraints are specified as three-tuples $\{\mathcal{P}, \mathcal{T}_\mathcal{P}, \mathcal{C}_\mathcal{P}\}$, where $\mathcal{P}$ specifies the unique name of the path, $\mathcal{T}_\mathcal{P} = t_0, \ldots, t_n$ is the ordered set of tasks contained in $\mathcal{P}$, and $\mathcal{C}_\mathcal{P}$ specifies the constraint for $\mathcal{P}$. These tuples are broadcasted to all analysis instances, which store paths and constraints relevant to them. Once, $\mathcal{T}_\mathcal{P}$ is known in the system, constraints may be updated using the tuple $\{\mathcal{P}, \mathcal{C}_\mathcal{P}\}$ to reduce communication overhead.

To be able to detect violations of path latency constraints, each analysis instance must be able to compute the total latency along a given path. We propose the following algorithm to compute this latency.

Whenever an analysis instance needs to compute the latency along a specific path, it sends a request for path latency calculation along $\mathcal{P}$ towards $t_0 \in \mathcal{T}_\mathcal{P}$. From this source, a path latency discoverer is sent along the path towards its sink. This discoverer accumulates latencies along the path. Once, the discoverer reaches the end of the path, the latency calculation ends and the total path latency is stored in the discoverer. This information is then sent back along the path towards the source. This way, all analysis instances alongside the path get updated latency information. Algorithm 1 shows the pseudo-code for the algorithm applied to non-end tasks in the path. To avoid duplicate latency calculations, t_0 drops any calculation request between starting a discoverer and receiving the corresponding result.

Using the mechanisms described above, an acceptance test can be implemented. If new functionality is to be integrated into a system, its timing properties, i.e. core execution times, sizes of exchanged messages, etc., and constraints must a priori

Algorithm 1 Calculate Path Latency

INPUT: Name of path $\mathcal{P}$
Task $t_i \in \mathcal{T}_\mathcal{P}, i > 0 \wedge i < n$ for current latency accumulation.
State $s \in \{REQUEST, DISCOVER, BC\_RESULT\}$
Current latency l
if $s == REQUEST$ **then**
  forward request to t_{i-1}
else if $s == DISCOVER$ **then**
  $l \leftarrow l+$ latency imposed by t_i
  forward discoverer to t_{i+1}
else
  store l as current latency of $\mathcal{P}$
  forward result to t_{i-1}

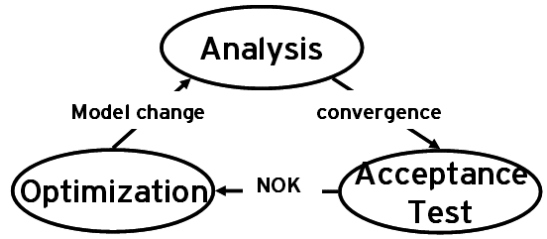

Fig. 2 - SELF-ADAPTATION MODUS

be specified in a format convertible to a SymTA/S compliant model. This description is integrated into the existing distributed model of the running system. After convergence of the online analysis, the new functionality can be accepted if (a) no resource is overloaded and (b) no path constraints are violated.

We propose that similar to the concept of "live analysis", an acceptance test is executed every time convergence of analysis has been detected. Each analysis instance runs the acceptance test by checking all contained resources for overload situations and querying the relevant paths for their metrics and comparing them to given constraints. In case a resource is overloaded or a constraint is violated, a NOK (not OK)-message is broadcasted throughout the system to indicate that the current configuration cannot be accepted.

These messages can be used to trigger self-protecting or self-adapting algorithms to resolve the detected problems. These algorithms can range from denying a proposed update over graceful degradation to self-optimization as described below.

V. HEURISTIC ONLINE OPTIMIZATION

In this section we present a heuristic online optimization algorithm which can be used in conjunction with the acceptance test introduced in Section IV to increase the acceptance rate for the integration of new functionality into a given organic system. In Section VI this algorithm is used to successfully integrate new functionality into the example system given in Figure 1.

Note that the presented algorithm is tailored for optimizing the priority assignment of tasks or messages mapped on priority scheduled resources, i.e. static priority preemptive scheduled CPUs and CAN arbitrated buses. However, we are currently working on extending it to various additional scheduling strategies, including time-slice based schedulers (e.g. TDMA, Round Robin), and automotive RTOSes (e.g. $ERCOS^{EK}$).

Algorithm 2 optimizes the priorities of tasks running on a single priority scheduled resource. In order to realize a system-level priority optimization, each resource needs to run its own instance of the optimization algorithm.

During run-time of the organic system, the system can be set into the *self-adaptation modus* (Figure 2) by broadcasting a message through the system.

In self-adaptation modus the SymTA/S instances running on the resources in the system react to NOK-messages of the acceptance test (Section IV) by triggering an online optimization step. Note that the necessary input data for Algorithm 2, i.e. path latencies etc., can be collected during the execution of the acceptance test.

During an online optimization step model parameters (i.e. task and message priorities) are modified, which, in turn, leads to the triggering of the live analysis. Once the live analysis has converged, a new acceptance test is performed. For the case that the system still violates constraints, it broadcasts a NOK-message, which again triggers an online optimization step. This analysis-optimization loop is repeated until a working system configuration is found, i.e. no NOK message occurs for a sufficiently long amount of time (time out), or until a certain number of unsuccessful optimization steps have been performed.

Algorithm 2 k-th Online Resource Optimization Step

INPUT: CPU $\mathcal{C}$, tasks $\mathsf{t}_1 \ldots \mathsf{t}_n$ mapped on $\mathcal{C}$ with response times $\mathsf{r}_1 \ldots \mathsf{r}_n$, constrained paths $\mathcal{P}_1 \ldots \mathcal{P}_m$ with deadlines $\mathcal{D}_1 \ldots \mathcal{D}_m$ and end-to-end response times $\mathcal{R}_1 \ldots \mathcal{R}_m$,
histories $\mathcal{H}_1 = \{\mathcal{H}_1^1, \ldots, \mathcal{H}_1^{k-1}\}, \ldots, \mathcal{H}_n = \{\mathcal{H}_n^1, \ldots, \mathcal{H}_n^{k-1}\}$,
where $\mathcal{H}_i^j = \{[\mathsf{pls}^j_{i,x_i^1}, \mathsf{rpl}^j_{x_i^1}], \ldots, [\mathsf{pls}^j_{i,x_i^{s_i}}, \mathsf{rpl}^j_{x_i^{s_i}}]\}$
history depth h_{depth}
OUTPUT: Priority assignment for $\mathsf{t}_1 \ldots \mathsf{t}_n$
1: **for** $(i = 1$ to $m)$ **do**
2: $\mathsf{rpl}_i^k = \frac{\mathcal{R}_i - \mathcal{D}_i}{\mathcal{D}_i}$
3: **for** $(j = 1$ to $n)$ **do**
4: **if** $(\mathsf{t}_j \in \mathcal{P}_i)$ **then**
5: $\mathsf{pls}^k_{j,i} = \frac{\mathsf{r}_j}{\mathcal{R}_i}$
6: add $[\mathsf{pls}^k_{j,i}, \mathsf{rpl}_i^k]$ to $\mathcal{H}_j^k$
7: **for** $(i = 1$ to $n)$ **do**
8: $\mathsf{relevance}_i = 1$
9: **for** $(j = k$ downto $k - \mathsf{h}_{depth} + 1)$ **do**
10: **for** $(q = 1$ to $s_i)$ **do**
11: $\mathsf{relevance}_i = \mathsf{relevance}_i \times (1 + \mathsf{pls}^j_{i,x_i^q}) \times (1 + \mathsf{rpl}^j_{x_i^q})$
12: assign highest priority to process with highest relevance, etc.

Algorithm 2 heuristically determines the "importance" of each task on the considered resource and distributes the priorities accordingly.

The algorithm iterates over all global paths known by the SymTA/S instance running on the optimized resource.

For each path the algorithm calculates the *relative path lateness (rpl)*, denoting the percentual lateness, i.e. the percentage by which the path deadline is violated (line 2). Note that negative lateness means that the response time of the path is inferior to its deadline. Afterwards, the algorithm determines for each tasks contained in the current path the *process latency share (pls)*, denoting its response time share in the overall path latency (lines $3-6$). Note that the tuples (*pls*,*rpl*) for each path-task pair are stored in the history of the current optimization step (line 6).

Afterwards, the relevances for the tasks mapped on the considered resource are determined (lines $7-11$). For this

purpose, the algorithm considers the history data of the last h_{depth} optimization steps, if existent. We observe, that both high *rpl* and high *pls* values lead to a task relevance increase. Note that negative *rpl* values for a path, i.e. the path holds its deadline, decrease the task relevance.

Finally, the algorithm assigns the priorities in a relevance monotonic fashion, i.e. the process with the highest relevance is assigned the highest priority, etc.

VI. Case study

In this section we take the example system shown in Figure 1 as case study to demonstrate the algorithms proposed in the previous sections.

Suppose the filtering application consisting of the path $S_{in} \rightarrow S_{out}$ shown in figure 1(b) is to be integrated into the system. In order to not compromise any timing properties of the applications already running on the system, the tasks of the application are assigned the lowest priorities on the respective resources. This is already reflected in Table I.

Integration of the additional functionality into the model triggers online analysis as previously described. In this case, online analysis starts on the processors, on which new functionality is integrated - the ARM and DSP processor. Changed timing behaviour on these resources triggers renewed analysis on the CAN bus, which in turn triggers an analysis on all connected Processors that read data from the bus. This loop will continue until convergence of analysis. Note that the CAN bus analysis can be triggered each time an attached processor has been analysed. A topological sort of the system architecture shows however, that it would be sufficient to first analyse all processors connected to the bus and then analyse the CAN bus once. In a worst case scenario, this could result in the CAN bus being analysed four times, when only one analysis run is needed. This topological information is being exploited during offline analysis, our distributed framework however does not yet gather any topological information and thus has an overhead in analysis runs as described before. Experiments show, that "live analysis" analyses the CAN bus about twice as often as offline analysis for the setup shown in figure 1.

Once live analysis has converged, an acceptance test is triggered on each resource, detecting that timing constraints of the new application as well as a constraint of the old application are violated (see Table II(b)), hence the acceptance test fails and broadcasts a NOK-message through the system. In this case, all analysis instances will broadcast a NOK message, since all contain at least one task that is part of a path whose constraint is violated.

If the system is in self-adaptation mode, this triggers an online optimization step, which may find a different configuration of the system for which no path constraints are violated. In this case, optimization yielded the following priority assignment leading to a working system after 3 optimization steps. CAN: $C_5 > C_4 > C_0 > C_1 > C_2 > C_3$, ARM: $T_8 > T_6 > T_1$, DSP: $T_7 > T_4$, PPC: $T_2 > T_5$, μC: $T_0 > T_3$.

VII. Conclusion

In this paper we presented a new methodology introducing self-awareness and self-adaptation with respect to real-time constraints into complex distributed embedded systems. The ambition of our approach is to increase system robustness, flexibility, and extensibility without explicit engineering effort. We motivated our approach by means of a small but realistic case study and demonstrated that online performance analysis coupled with online optimization can increase the acceptance rate for software updates in embedded systems. Similarly, a greater robustness to subsystem failures can be achieved.

Path	*wc lat.*	*OK*
$Sens \rightarrow Act$	707.4	$\surd$
$Cam \rightarrow V_{out}$	392.8	$\surd$
$S_{in} \rightarrow S_{out}$	n.a.	n.a.

(a) latencies before integration

Path	*wc lat.*	*OK*
$Sens \rightarrow Act$	861.9	X
$Cam \rightarrow V_{out}$	484.7	$\surd$
$S_{in} \rightarrow S_{out}$	2639.9	X

(b) latencies before optimization

Path	*wc lat.*	*OK*
$Sens \rightarrow Act$	795.8	$\surd$
$Cam \rightarrow V_{out}$	1067.7	$\surd$
$S_{in} \rightarrow S_{out}$	509.1	$\surd$

(c) latencies after optimization

TABLE II - Latencies before and after integration and optimization

References

[1] E. Bini, G. C. Buttazzo, and G. M. Buttazzo. Rate monotonic scheduling: The hyperbolic bound. *IEEE Transactions on Computers*, 52(7):933–942, July 2003.

[2] E. Bini, M. Di Natale, and G. C. Buttazzo. Sensitivity analysis for fixed-priority real-time systems. In *Proc. of the Euromicro Conference on Real-Time Systems (ECRTS)*, Dresden, Germany, July 2006.

[3] S. Chakraborty, S. Künzli, and L. Thiele. A general framework for analysing system properties in platform-based embedded system designs. In *Proc. of the IEEE/ACM Design, Automation and Test in Europe Conference (DATE)*, Munich, Germany, 2003.

[4] S. Funk, J. Goossens, and S. Baruah. On-line scheduling on uniform multiprocessors. In *Proc. of the 22nd IEEE Real-Time Systems Symposium (RTSS)*, London, UK, 2001.

[5] A. Hamann, R. Henia, M. Jersak, R. Racu, K. Richter, and R. Ernst. SymTA/S - Symbolic Timing Analysis for Systems. *http://www.symta.org/*.

[6] A. Hamann, M. Jersak, K. Richter, and R. Ernst. A framework for modular analysis and exploration of heterogeneous embedded systems. *Real-Time Systems Journal*, 33(1-3):101–137, July 2006.

[7] R. Henia, A. Hamann, M. Jersak, R. Racu, K. Richter, and R. Ernst. System level performance analysis - the SymTA/S approach. *IEE Proceedings Computers and Digital Techniques*, 152(2):148–166, March 2005.

[8] Paul Horn. Autonomic computing: Ibm's perspective on the state of information technology. *http://www.research.ibm.com/autonomic/manifesto/autonomic_computing.pdf*, October 2001.

[9] David B Johnson and David A Maltz. Dynamic source routing in ad hoc wireless networks. *Mobile Computing*, 353, 1996.

[10] Christian Mueller-Schloer. Organic computing - on the feasibility of controlled emergence. In *IEEE/ACM/IFIP International Conference on Hardware/Software Codesing and System Synthesis (CODES + ISSS 2004)*, 2004.

[11] C. Perkins, E. Belding-Royer, and S. Das. Ad hoc On-Demand Distance Vector (AODV) Routing. RFC 3561 (Experimental), July 2003.

[12] R. Racu, A. Hamann, and R. Ernst. A formal approach to multi-dimensional sensitivity analysis of embedded real-time systems. In *Proc. of the Euromicro Conference on Real-Time Systems (ECRTS)*, Dresden, Germany, July 2006.

[13] K. Richter. *Compositional Performance Analysis*. PhD thesis, Technical University of Braunschweig, 2004.

[14] Thilo Streichert, Dirk Koch, Christian Haubelt, and Jrgen Teich. Modeling and design of fault-tolerant and self-adaptive reconfigurable networked embedded systems. *EURASIP Journal on Embedded Systems, Special Issue on Field-Programmable Gate Arrays in Embedded Systems.*, 2006.

[15] David Tennenhouse. Proactive computing. *Commun. ACM*, 43(5):43–50, 2000.