Self-Configuration in Hard Realtime Systems

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

***Abstract*—We present a runtime solution to self-configuration of systems that are subject to hard realtime constraints. The presented approach builds on an admission control scheme, that verifies the implications of a configuration change on system timing, prior to the actual reconfiguration process. Based on this verification process, the self-configuration finds feasible priority assignments for otherwise unacceptable applications, considering static priority preemptive (SPP) scheduling and end-to-end path latencies.**

I. Introduction

Many embedded systems evolve over their lifetime - be it through updates or autonomous adaptation to their environment. If these systems have to adhere to hard constraints, it becomes imperative that feasibility of adaptations is verified before applying them in the system. In [3] we have presented the EPOC framework, which implements a model-based admission control scheme that performs a timing verification within the system to provide hard realtime systems with self-protection capabilities against infeasible adaptations. In this demonstration we present an extension to this admission control scheme that allows to autonomously find scheduling parameters under which it is possible to accommodate adaptations, that would have otherwise been rejected.

First we introduce compositional performance analysis theory, which forms the algorithmic basis of the timing analysis, within the admission controller. Then we recapitulate the EPOC architecture. We will then outline the architectural approach and the algorithmic basis of the self-configuration service. Finally we will elaborate on the demonstrator setup and the properties of our framework.

II. In-system Performance Analysis

The model-based timing verification within the EPOC framework builds on compositional performance analysis [1]. Systems are modeled as task graphs which are mapped on computational and communication resources. A task is activated due to activating events. Furthermore, 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 task response times using task set descriptions and activating event models as input data. 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.

Compositional performance analysis solves the global system-level scheduling analysis problem by alternating local scheduling analyses and *event model propagation* along *event streams* connecting functionally dependant tasks.

Global system level performance analysis yields relevant timing data such as task response times and end-to-end latencies for task chains possibly spanning multiple processors.

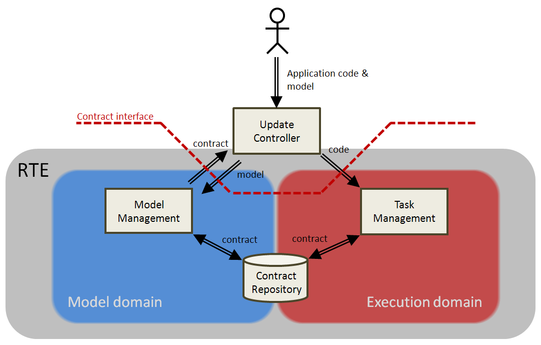


Figure 1. EPOC-RTE Contracting Architecture

The global system-level scheduling analysis can efficiently be implemented as a distributed algorithm [4]. The EPOC framework implemented on the demonstrator uses such a distributed implementation of system-level performance analysis to decide about feasibility of a configuration at run-time.

III. Runtime Environment

In order to use such a model-based analysis as a run-time feasibility evaluator, the behavior and demands of the software as well as the capabilities of the platform must be expressible in an efficient way. We use *contracts* as description, where the requesting entity specifies its own behavior/configuration and receives assertions w.r.t. its constraints. We evaluate contract requests at run-time using in-system performance analysis. An application or adaptation is only admitted to the system, if analysis shows that all contracts are satisfied. Thus the system only transitions between provenly performance-safe configurations.

Application contracts consist of a performance characterization of the application as guarantee for the RTE and a set of constraints the application assumes to adhere to. In our system this is a description of the task-graph that constitutes the application, of the worst-case timing behavior and of temporal constraints (end-to-end latencies). If the performance evaluation of a set of contracts yields that all contracts can be met, the RTE asserts that all applications comply to their constraints given they behave as described in the contract request.

The architecture of the EPOC-RTE, as shown in figure 1, is separated into two domains - the *Model Domain* (left) and the *Execution Domain* (right). While the former negotiates contracts based on the provided model the latter enforces the parameter settings of the model. The Contract Repository acts as an interface between both domains. This architecture provides the advantage of decoupling analysis of system configurations from execution of accepted applications. A further advantage is that system configurations can be evaluated and explored in the model domain without the necessity of executing the configuration on the actual system.

IV. Self-configuration Approach

We will now look at the Model Domain, and its extension for self-configuration, in more detail. Figure 2 depicts the framework architecture within the Model Domain.

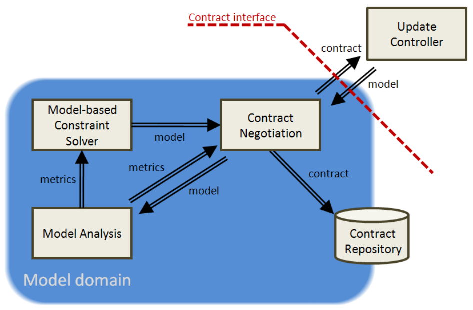


Figure 2. Refined view of the Model Domain

New or updated applications are presented to the framework through the *Contract Interface* using the Update Controller. The *Contract Negotiation* component inserts the received model information into the *Model Analysis* [4] to determine feasibility of the system change. In case the system change can be accepted, the contract is stored in the *Contract Repository* and the result reported to the Update Controller. The Execution Domain then enforces the change.

If however an application cannot be accepted with the provided parameters, the RTE autonomously tries to find a configuration under which all constraints are satisfied. To accomplish this, the analysis component is extended by a *Model-based Constraint Solver* [2]. In case an infeasible update is detected, Model Analysis reports timing metrics to the constraint solver, which creates a new configuration based on the metric δ, which denotes a task's responsibility for a path latency violation (ω denotes the task's worst-case response time, λ the worst-case path latency and χ the path latency constraint).

$$\delta \quad = \quad \max\left(0, \frac{\omega}{\lambda} * (\lambda - \chi)\right) \qquad (1)$$

Scheduling priorities are assigned in descending order of δ. The new configuration is again presented to Contract Negotiation for evaluation. This control loop within the Model Domain is executed until a feasible configuration is found or until a maximum number of iterations is reached.

V. Demonstrator Setup

Our demonstrator consists of a hardware setup, performing a control task, which resembles a timing sensitive application. We demonstrate that the runtime environment, as described above, detects a possible violation of timing constraints upon insertion of a music streaming application. It then autonomously determines a priority assignment under which all constraints of both applications are met, thus allowing parallel execution.

The seesaw inspired setup is depicted in fig. 3. The beam in the middle is mounted movable at both ends on the surrounding frame. The position of each end can be controlled by separate stepper motors. One end is controlled by a user definable disturbance while the other end is positioned by a closed-loop control application. The track accommodates a ball which is to be balanced in the center position at all times. To fulfill this task one controller is equipped with an angle sensor and an array of IR photosensors.

The stepper motors are each connected to a controller board which feature the proposed framework (fig. 4a). The disturbance application (D_1) resides on the right controller board and operates the locally connected stepper motor. The control application (C_{Ctrl}, C_{Comm}, C_{Sensor}) is distributed over both boards. The sensor task acquires data from the sensors and transmits it via the CAN-Bus to the controller task which positions the second stepper motor accordingly. The latency from the sensor task to the controller task is constrained to ensure correct operation of the control loop.

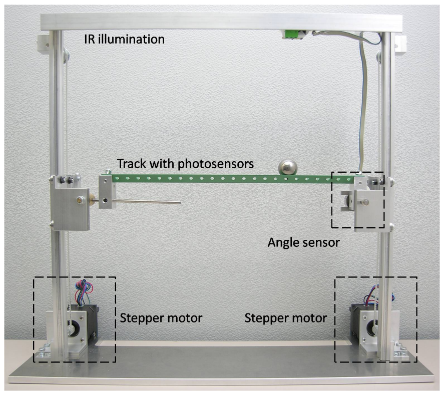
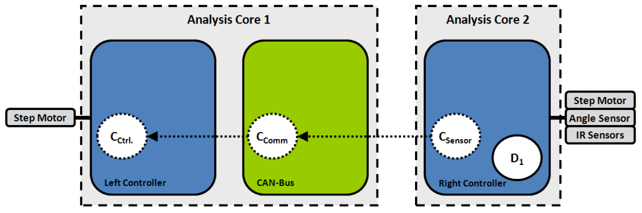
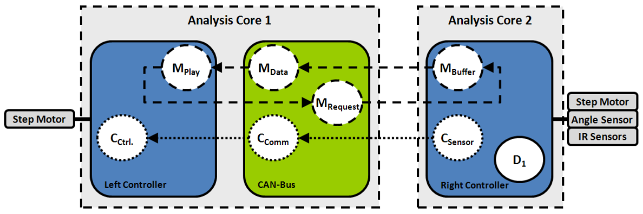


Figure 3. Mechanical setup of the demonstrator



(a) System setup without update



(b) System setup with update

Figure 4. System setup of the demonstrator

VI. Demonstrator Analysis

When the user tries to insert the music application (M_{Buffer}, M_{Data}, M_{Play}, $M_{Request}$) (fig. 4b) the framework performs a distributed performance analysis to determine whether the control and music application can be executed quickly enough to fulfill all timing constraints. If required it autonomously reassigns execution priorities, to solve the conflicts.

VII. Conclusion

We demonstrate a functional runtime environment that enables runtime performance verification in hard realtime embedded system. Timing metrics from the verification algorithm are used to autonomously reassign execution priorities. Thus providing self-configuration capabilities to the system.

References

[1] R. Henia, A. Hamann, M. Jersak, R. Racu, K. Richter, and R. Ernst, "System level performance analysis - the symta/s approach," *Computers and Digital Techniques, IEE Proc.* -, vol. 152, pp. 148–166, 2005.

[2] M. Neukirchner, S. Stein, and R. Ernst, "A lazy algorithm for distributed priority assignment in real-time systems," in *Proc. of 2nd IEEE Workshop on Self-Organizing Real-Time Systems (SORT)*, 2011.

[3] M. Neukirchner, S. Stein, H. Schrom, and R. Ernst, "A software Update Service with Self-Protection Capabilities," in *Proc. of the conf. on Design, Automation and Test in Europe (DATE)*, 2010.

[4] S. Stein, A. Hamann, and R. Ernst, "Real-time property verification in organic computing systems," in *Second Int'l. Symp. on Leveraging Applications of Formal Methods, Verification and Validation*, 2006.