Consistency Challenges in Self-Organizing Distributed Hard Real-Time Systems

Steffen Stein, Moritz Neukirchner, Harald Schrom, Rolf Ernst

Institute for Computer and Network Engineering (IDA)
Technische Universität Braunschweig
Braunschweig, Germany
Email: stein|neukirchner|schrom|ernst@ida.ing.tu-bs.de

***Abstract*—Allowing real-time systems to autonomously evolve or self-organize during their life-time poses challenges on guidance of such a process. Hard real-time systems must never break their timing constraints even if undergoing a change in configuration. We propose to enhance future real-time systems with an in-system model-based timing analysis engine capable of deciding whether a configuration is feasible to be executed. This engine is complemented by a formal procedure guiding system evolution.**

The distributed implementation of a runtime environment (RTE) implementing this procedure imposes two key questions of consistency: How do we ensure model consistency across the distributed system and how do we ensure consistency of the actual system behavior with the model?

We present a synchronization protocol solving the model consistency issues and provide a discussion on implications of different mode-change protocols on consistency of the system with its model.

I. Introduction

One of the largest challenges in 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 upperclass automobiles can be ordered in up to 2^{25} configurations with possible effects on the electric system [FHS05]. This is only possible because of software flexibility, but it turns integration into a hard task as all of these configurations need to be analysed for feasibility before shipping. Also, it has to be repeated for each update or other change in configuration to be applied to an automobile.

A remedy to this problem is to move the verification process into the embedded system - in this case the car. If the system is aware of its key performance data, it could protect itself from maliciuous updates that would compromise the correct timing of existing applications. In order to do this, only one - the upcoming - configuration needs to be analyzed for each update. We implemented a prototype capable of protecting itself against updates breaking timing contracts in the EPOC project [NSSE10].

When constructing real-time systems that are allowed to autonomously modify ("self-organize") themselves, one must ensure that they will never break their real-time constraints as a result of this self-organization. We believe that the EPOC architecture [NSSE10] is an enabler for the implementation of such systems as it allows to formally verify real-time properties within the system and thus deminishes the need for a-priori analysis of all configurations a self-organizing real-time system may pass through during its life time.

The EPOC architecture as introduced in [NSSE10] verifies system configurations based on a formal timing model and manages software applications distributedly without a central entity. This ensures scalability and distributes the load of management and analysis evenly throughout the system.

As no system component has an overall view of the system state, and no central entity globally coordinates an ongoing modification, several data and state consistency issues arise. As first aspect we will discuss these and present a synchronization protocol solving them.

As a second aspect, we will show that it is not sufficient to analyze the behavior of a system before and after a change in software. Transient configurations as will appear during configuration transitions can severly affect the timing properties of the upcoming configuration. Thus, the algorithms and protocols used to transition between successive configurations must be considered. We discuss possible strategies for configuration transitions on an abstract level to highlight the potential timing issues associated with them. We will start with a motivating example in the following section.

II. Motivating example

Take the system shown in figure 1 as an example. It consists of three processors interconnected by two buses and thus resembles a typical gateway setup. Assume that this system is currently running a control application spanning all three processors, as the necessary sensors are attached to CPU 1 and the actuators are attached to CPU 3. For stable operation of the control loop, constraints on the maximum path latency as well as the maximum allowed jitter exist. Assume that adherence to these constraints has been shown during the development process using state of the art tools. In the remainder of the paper, we refer to this application as the old or o-application.

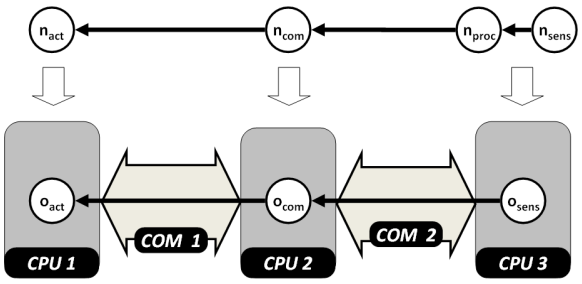


Figure 1: Example System

In this example, assume that a second application (new or n-application) is to be added to the system after deployment which will run on the same platform, again spanning all three processors. The application consists of four tasks and is shown in the top of figure 1. This application, similar to the already existing one, must be guaranteed a constrained latency as well as activation jitter during operation. The question of whether the update is feasible cannot be answered without knowledge of all other applications running on the system as shared usage of resources, such as processors and buses lead to complex interdependencies in the timing behavior of the applications.

Using a state-of-the art development process, the update can only be installed on the embedded system after extensive lab testing, this means returning to the verification stage of the system development process after deployment of the actual system, which is costly. Our proposed remedy to this problem is to enable the system to autonomously decide whether an update is suitable for execution. In order to enable the embedded system to perform this decision, we propose to implement an in-system model-based performance evaluation engine.

Adding tasks or applications at run time is just one of many possible mechanisms to achieve self-organization or adaptivity in (not only) real-time systems. Other mechanisms include migration of tasks between processors, adaptation of scheduling parameters such as task priorities or variation of hardware parameters such as processor frequency, which all have an impact on real-time properties of the system. An RTE of a self-organizing real-time system (and thus the principles presented in this paper) should consider all these mechanisms.

In this paper, we will shortly introduce the main concepts and the architecture of the EPOC-RTE. We will then go into more detail on the distribution of the RTE over the embedded systems and consistency issues arising from this distribution. Section VII describes a synchronization protocol we use to ensure consistent states within the RTE. The section thereafter goes into detail on timing consistency issues encountered when transitioning between different system configurations.

III. Contracting

By contracting, we understand that applications provide information on their behavior and needs to the system, which, before starting an application, ensures that given all applications adhere to their description, they will be able to meet their constraints. We refer to this agreement between application and system as a contract. The EPOC RTE relies on the concept of *in-system contracting* to allow save updates w.r.t. timing properties.

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. In line with nomenclature common in the area of performance analysis, we will refer to processors and buses as computational and communication *resources*, respectively.

We assume that the EPOC RTE ([NSSE10]) is running on each processor within this platform. 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.

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 and the applications executed within these RTE instances. 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. Furthermore task specific information is available *locally* if it is only known to the resource that hosts that particular task.

Changes in a real-time system such as adding or removing tasks may affect parts of the system that are not directly associated with the transition. In performance analysis literature, this effect is known as a *non-functional dependency*. In this paper, we refer to the set of resources that is affected by the current transition as the *effect horizon* of the current transition. The effect horizon contains at least all resources, that are directly affected by the update. Depending on the system configuration it may include additional resources, that are affected due to non-functional dependencies.

In order to be able to verify system properties at runtime, contract information has to be provided along with an application. A *contract* contains a description of the application (e.g. task graph), a set of prerequisites of the application and a set of guarantees it can give in turn. In our context of performance verification, application contracts consist of a

performance characterization of the application in isolation as guarantee for the RTE (i.e. worst-case execution times (WCET)), a set of constraints the application assumes to adhere to (i.e. constraints on worst-case path latencies) and a description of communication dependencies (i.e. task graph). If a contract is accepted, the RTE guarantees that all applications comply to their constraints, given they do behave as described in the contract. For a more detailed description of the contracts please refer to [NSSE10].

IV. Architectural Approach

In the this section we will recapitulate the EPOC architecture and explain the associated contract-based update protocol, that allows for in-system verification of timing properties, with the example system shown in figure 1.

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, as contained in the contract information, 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.

The core component of the Model Domain is the Model Analysis. It performs the feasibility check - i.e. a distributed performance analysis algorithm ([SHE06]). Models, as required by the analysis, are supplied by the Contract Negotiation component, which accepts contract requests 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.

The Execution Domain, as depicted on the right hand side of figure 2, enforces application contracts by setting execution parameters of applications. 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). The Execution Domain is also capable of starting and stopping as well as migrating tasks at run time.

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 controlled according to their model.

Note that the EPOC framework is a distributed RTE. Thus the shown components exist on every resource and cooperate to fulfill their respective tasks and thus act as cooperative software agents.

V. Degree of Distribution

In the following we will focus on the degree of distribution of the model and the applications. Some of the consistency issues, that will be discussed in sections VII and VIII, arise immediately from this degree of distribution.

	local	connected	unconnected
WCET	✓	-	-
Mapping	✓	✓	-
Path latency	✓	-	-

Table I: Summary of visibility of tasks

If model and applications are distributed in the system to a high degree, additional coordination between participating processors is required and the view each resource has of the system setup is restricted. However, a high degree of distribution may significantly reduce memory usage and improve scalability, as data is only present where actually needed.

As already indicated in the previous section, the model of the system configuration is held distributedly. Every resources' local Contract Repository only contains the information, that is needed at that particular resource. Contract information can be partitioned in application and task specific information. Application specific information contains data relevant for the entire application (e.g. constraints on path latencies), while task specific information only describes properties of one particular task (e.g. communication dependencies of that task, WCET). All task specific information is only available locally, i.e. only at the processors that actually executes the specific tasks. Application specific information is available application-wide.

Table I summarizes the visibility a resource has of the system model. Resources are aware of their local tasks and the corresponding communication partners (connected tasks), though cannot access the task specific information (e.g. WCET) of those communication partners (as indicated by the dashed lines). Additionally, resources do not know the extend of an application beyond the communication partners of their tasks.

Again, consider the example system as depicted in figure 1. Figure 3 shows the resources' view of the system model. E.g. CPU 1 has knowledge of all contract information of n_{act} and is aware that a task n_{com}, that is communicating with n_{act}, exists on CPU 2. It has no information though about the existence of tasks n_{proc} and n_{sens}. Furthermore CPU 1 is aware, that n_{act} is part of a latency-constrained path. The Model Analysis component provides information on the path latency on all resources, that host at least one task of that path.

The degree of distribution within the execution domain is comparable. The program code of all tasks is only stored locally. Communication between tasks is established via a runtime-configurable communication middleware within the RTE.

As a result of the distribution of model and code and the lack of a central entity within the RTE, keeping the model and the execution of the application in a consistent state, becomes a major challenge. In the following section

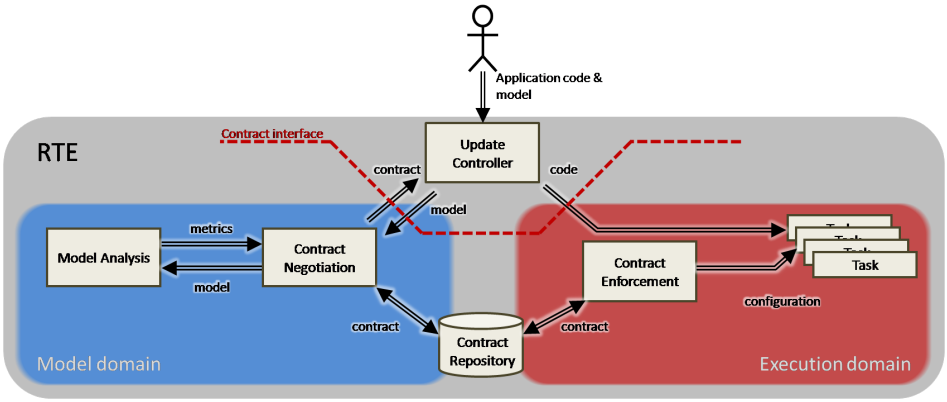


Figure 2: Framework Architecture

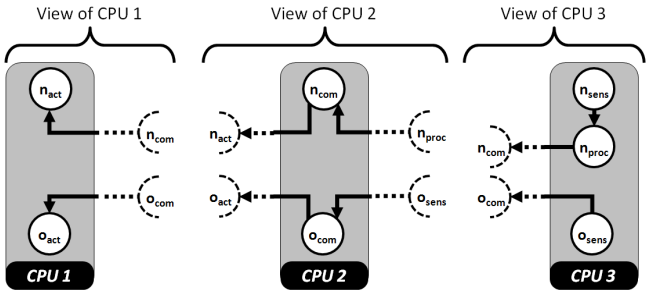


Figure 3: Resources' view of the system configuration

we will outline the update process of the EPOC architecture and refine these challenges. We will then focus on these consistency issues throughout rest of this paper.

VI. The Update process

In this section we will delineate the process of adding an application on the EPOC architecture. Figure 2 will serve as reference for the framework components, while figure 4 depicts the sequence chart for the update.

Updates are provided by the Update Controller. For each update one Update Controller controls the distribution of the update - potentially a different one for any update. This *Master* Update Controller stores all information about the update. This includes the update description in the form of contracts as well as the program code to be executed if the update is accepted. In the presence of gateway architectures all Update Controllers cooperate for routing.

Consider the example system shown in figure 1. The o-application, consisting of a sensor task o_{sens}, a communication task o_{com} on the gateway resource CPU 2 and an actuator task o_{act}, is already present in the system. The n-application shall be added to the system configuration with a mapping as indicated by the arrows. Let the Master Update Controller reside on CPU 3.

In a first step (1) the Update Controller has to transmit the contract information of task n_{com} to CPU 2 and of task n_{act} over CPU 2 to CPU 1. On each resource the Contract Negotiation will then store the corresponding contract information in the local Contract Repository (2) and insert the model into the Model Analysis component (3).

As no assumptions are being made on platform architecture and communication media, no statement about the arrival times of contract information at the corresponding resource, as distributed by the Update Controller, can be made. Thus no resource knows when all contract information for a new or updated application has been received at all affected resources. If a resource would start analysis immediately after reception of its updated contract information, some parts of the model on other resources might still be outdated. In this case the analysis results would not reflect a worst-case estimation of the system behavior and thus no reliable statement on system feasibility could be made. As a result a synchronization mechanism (4) is required to ensure model consistency.

After analysis (5) has been performed cooperatively by all three processors, the performance metrics are evaluated (6) locally at each resource. As all resources only have knowledge of timing metrics of tasks and paths, that they host, they can only evaluate the constraints for those tasks and paths. Thus another synchronization step (7) is required, to gather analysis results to be able to determine feasibility of the system configuration.

If the n-application can be accepted, the Master Update Controller on CPU 3 is notified (8) and the update can be executed in the execution domain. The program code of tasks n_{com} and n_{act} is transferred to CPU 2 and CPU 1 respectively (9). In another synchronization step (10) the completion of all code transfers is assured. At this point in time all affected RTE instances have the necessary data to perform the transition to the next configuration. To actually enforce the new configuration a specific protocol is needed (11). In section VIII, we discuss several possible strategies for such protocols and their implications on system timing. Once the system is running the next configuration, an application wide synchronization (12) marks the end of the

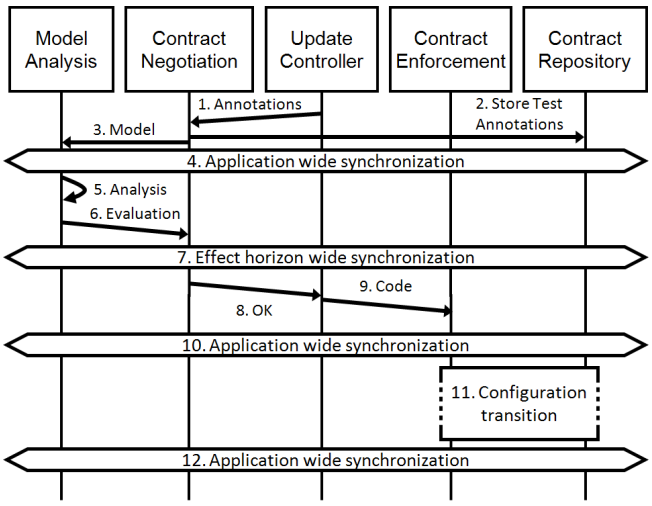


Figure 4: Update Protocol

update process.

VII. Enforcing Model Consistency

In this section we will address the issues of model and analysis consistency as outlined in section VI. We will define the requirements on synchrony and distribution of possible synchronization algorithms.

As indicated in figure 4 synchronization of model information is required application-wide, as the remainder of the system remains unchanged during the update of one application in steps 4, 10 and 12, and effect-horizon wide during step 7. We first elaborate on application-wide synchronization to then extend the approach to also cover the effect-horizon.

Since resources only have a limited view of the applications (section V), no resource knows all resources that have to be included in such an application-wide synchronization. Additionally we cannot assume a complete interconnect among resources, as gateway architectures are common in the domain of real-time systems (e.g. automotive platforms).

The system model in the model domain is only consistent, if all resources that are affected by the update, have received the new contract information (the same holds true for the completion of analysis and analysis results). In other words, every affected resource needs to perform a certain processing step, before any of the affected resources may proceed to the next processing step. No tight requirements are made on the time needed for this synchronization, as processing of model information is not timing critical. Instead resource usage and degree of distribution are of main concern. As real-time embedded systems are oftentimes very limited on resources (e.g. automotive platforms), especially concerning communication media and memory, a synchronization mechanism should have minimal implications on messages sent and memory consumption. Furthermore, as the EPOC framework has no central entity, also the synchronization mechanism should be distributed. At the same time, since applications can change over time through updates, it should be adaptive, to be able to handle e.g. migration of tasks to other resources.

A. Barrier synchronization

An approach very well-suited to the problem described, is known from the domain of *parallel computing*. In barrier synchronization ([Bro86], [GH89], [Lee90], [YK98]) a process enters a barrier, after it has completed all its computations. The processor may only proceed to execute, when all processes to be synchronized have entered the barrier. Implementations differ in communication structure between processors (e.g. master-slave, tree), complexity and platform requirements.

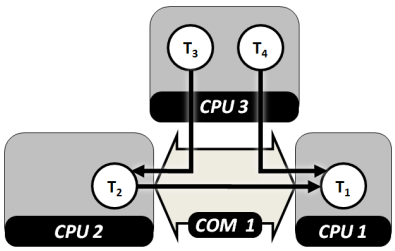
The reflected-tree barrier synchronization, presented in [Lee90], uses a tree communication structure between processors. In an *announcement* phase the leaves announce their arrival in the barrier towards the root. Nodes only forward the message, when they themselves arrive at the barrier and all leaf-sided nodes have announced their arrival. After arrival of all anouncements at the root of the tree, it notifies all nodes with a *notification* wave of successful synchronization. This algorithm can even be executed on platforms that do not have a complete interconnnect (e.g. gateway architectures), as only processors that are connected in the tree have to be able to communicate.

B. Extraction of communication trees

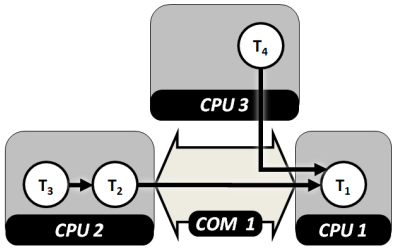
The reflected-tree algorithm is well-suited for the discussed synchronization requirements, due to its low message count and good scalability ([Lee90]). However it cannot be applied directly. The algorithm employs a tree communication structure between participating processors. However, the resources required to synchronize application-wide, may be different for every application and in the presence of task migration may even change for an application. Consider the following example.

Figure 5a shows a platform consisting of three processors and a communication bus. An application consisting of four tasks is mapped on the resources as depicted. In order to synchronize application-wide using the reflected-tree algorithm, all three processors would have to be included in the synchronization. Figure 6a shows a possible communication tree for this case, in which CPU 1 is the root.

Now, consider the case in which task T_3 is migrated to CPU 2 (figure 5b). Due to the limited model visibility, as discussed in section V, CPU 2 is not aware of any other task on CPU 3 after migration of Task T_3. Thus it does not know whether CPU 3 still has to be included in synchronizations. CPU 1 on the other hand, is aware of Task T_4 due to the communication with T_1. However, it does not see any of the

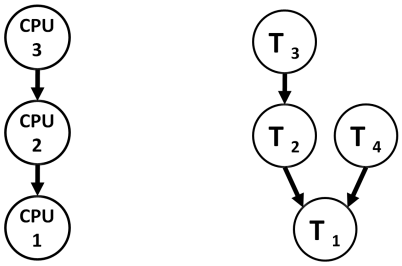


(a) Example system before migration



(b) Example system after migration of task T_3

Figure 5: Example system



(a) Resource-based communication tree

(b) Task-based communication tree

Figure 6: Tree communication structures

model changes due to the task migration. Thus it will not change synchronization behavior.

This scenario shows, that a resource-based synchronization tree cannot adapt to changing applications easily. Instead it may need to adapt the communication structure for the synchronization with every model change.

We propose, to extract the communication tree from the task graph of the application. The resources then communicate along the edges of that tree. Again, refer to the example from figure 5. A possible task-based tree for the given example system is depicted in figure 6b. Before migration CPU 3 would have to send one synchronization message for T_3 to CPU 2 and one for T_4 to CPU 1. After migration CPU 3 only has to send one message to CPU 1. On CPU 2 the message from T_3 to T_2 can be processed internally and only one message has to be send to CPU 1.

The tree information can be extracted from the task graph at design time and does not need to be changed in the presence of task migration. Additionally no resource requires knowledge of the entire tree. Instead for every task only the predecessors and the successor in the tree need to be known. The resources, that predecessors and successor are executed on, are already known from the local view of the system model due to communication dependencies (section V). The local information on the tree structure can easily be appended to the contract information and thus be updated if the task graph changes in the course of an application update.

C. *"Effect horizon"-wide synchronization*

After analysis the results of all model analysis components within the effect horizon have to be gathered. This may include resources, that do not contain any part of the changed application. However, this case can easily be covered with a minor extension to the task-based reflected-tree algorithm.

The model-based analysis components knows, when the timing behavior of an application changes due to an update of another application. Thus, it knows, when an application extends the effect horizon as a result of such a *non-functional dependency*. This locally available information can be exploited to adapt the synchronization tree.

The synchronization tree of the affected application can be linked as a leaf to the one of the updated application. Thus it will be included in the synchronization process. After completion, the linking can be removed. This way synchronization will only be performed on all resources, that are actually affected by an update.

Thus the reflecting-tree algorithm can be used for all synchronization steps in the update protocol (figure 4). It relies solely on local information, and imposes only minimal overhead on memory and processing. All required communication structures can be extracted at design-time and will only be linked for effect-horizon synchronization. Thus distributedly enforcing model consistency can be handled efficiently.

VIII. Configuration transitions

The second major consistency issue during updates of distributed real-time systems, as in the EPOC architecture, arises when the system transitions between two verified system configurations. This section addresses the problems that arise in step 11 of the update protocol (figure 4). When a system transitions between two configurations special precautions have to be taken, to ensure that the actual behavior of the system is consistent with the analyzed contracts. As we will show, timing behavior can easily deviate from the model during transitions if no special protocols are employed. The system designer has the options of either implementing special *mode-change protocols*, enhancing the analysis algorithms to reflect transitions, or a combination of both.

In the following, we will first give an overview of analysis approaches of such transitions and related mode-change protocols. Then we will define a classification of transition

scenarios and strategies for distributed systems and elaborate on their corresponding implications on timing and the design process. We do not provide a complete solution to the problem, but want to raise an awareness for the issue and indicate in which cases precautions have to be taken.

A. Analyzing configuration transitions

Existing research in schedulability analysis has shown, that even if both configurations - before and after a change in system configuration - have been proven feasible, the transition between both configurations can cause violation of timing constraints and thus needs explicit analysis. The protocols used to transition between different configurations in literature are referred to as *Mode-Change Protocols*. Most existing theoretical work in this domain focuses on single processor Mode-Change Protocols and their timing. A good overview can be found in [RC04]. The literature categorizes the tasks in a system undergoing a configuration transition in three sets:

1) *finished tasks*: The set of tasks that are present in the old, but not the new configuration. These tasks may have at most one activation during the transition phase.
2) *new tasks*: The set of tasks that are only present in the new configuration.
3) *unchanged tasks*: The set of tasks that are present in both configurations and remain unchanged in their parameters during the transitions.

The authors of [RC04] further introduce the set of tasks that are present in both configurations, but change their parameters, e.g. execution time or activating event model during the transition. We assume that these tasks are modeled using two tasks, one finished and one new, which is also common in literature.

Mode-Change Protocols are categorized by the way they treat these task sets. *Synchronous protocols* as opposed to *asynchronuous protocols* do not allow execution of finished and new tasks at the same time. Furthermore, protocols are discerned by the fact whether they accomodate unchanged tasks or not. Those that do are referred to as protocols *with periodicity* others are said to be *without periodicity*. It is very unlikely that there are no unchanged tasks during configuration transitions in complex distributed systems. Thus we assume all protocols mentioned from here on to be with periodicity.

Recent work [NGA09] also covers mode changes on multi-processors. The authors claim that their approach is the first published mode-change protocol dealing with multiple processors. It does however impose strong restrictions on properties of the processors, scheduling and task set. Namely the authors assume uniform multiprocessors platforms, a global (greedy) scheduler and no dependencies between tasks. Thus, the work is not applicable to the scenarios discussed here, as no global scheduler exist and tasks may have communication dependencies.

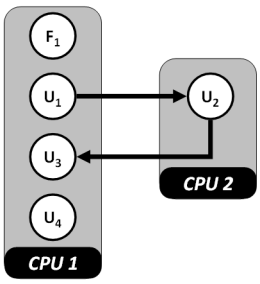


Figure 7: Echo-Effect Example

Uni-processor Mode-Change Protocols with dependant tasks have been discussed briefly in [HE07]. Here, it is observed, that unchanged tasks may experience different worst-case response times (WCRT) in the old and the new configuration as well as the transition. For communicating tasks, the difference in WCRT causes different potential communication patterns which in turn affect system timing.

Take the system in figure 7 as an example. It consists of five tasks mapped on two processors. We assume static priority preemptive scheduling on both processors and a priority assignment, such that $F_1 > U_1 > U_3 > U_4$. Furthermore assume that the system performs a mode change, with F_1 being a finished task and the tasks $U_1 - U_4$ being unchanged tasks. We consider the WCRT (and thus output jitter) of U_4. Clearly the WCRT of U_1, U_3 and U_4 decreases during the mode change and we can easily compute the WCRT for the old and new configuration using existing analysis techniques (e.g. [TBW94]). However, we do not know *when* we can safely assume the new WCRT.

Consider the instant in time the last activation of F_1 finishes. Clearly, after this point in time the new WCRT of U_1 and thus a more regular output pattern can safely be assumed. However, this effect has not yet arrived at the input of U_2 nor has it propagated to U_3. We are not aware of any analysis methodology that we can use to determine the necessary offset between the instance of the mode-change and the point in time at which we can safely assume the timing properties obtained when merely analyzing the new scenario. In general this problem can be much more complex than outlined here, as cyclic dependencies (functional and non-functional) between unchanged tasks may exist. We refer to these effects on timing affecting the system possibly for a long time after the cause (i.e. configuration transition) as *Echo-Effects*.

In [HE07], this problem is solved by assuming the worst case of the three states for system feasibility analysis. This constitutes a conservative approximation of the system's behavior before, after and during the transition. However, this does not constitute a feasible approach for real-time systems that undergo multiple configuration transitions, as the conservatism accumulates.

To express the effects of transitions on the timing behavior we use WCRT diagrams of unchanged tasks (e.g. figure 9).

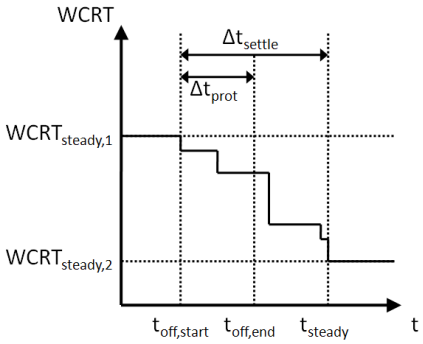


Figure 8: Settling behavior of an unchanged task for the scenario of stopping tasks

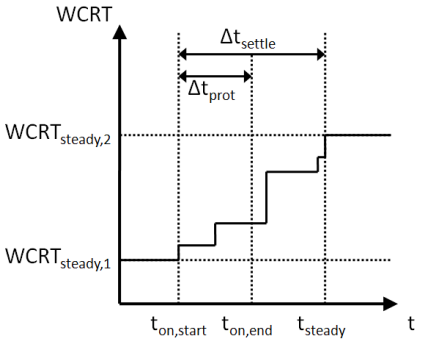


Figure 9: Settling behavior of an unchanged task for the scenario of starting tasks

The WCRT diagram depicts the worst-case response time a task may exhibit at time t. The steady state WCRTs $WCRT_{steady,1}$ of the old configuration and $WCRT_{steady,2}$ of the new configuration can be calculated by current performance analysis algorithms [TBW94].

A WCRT diagram of an unchanged task during a configuration transition that only stops tasks as discussed in the previous paragraphs can be seen in figure 8. The steady values of the WCRT can be computed for each configuration. Similarly, for any given protocol, handling the stopping of the tasks, we can compute the time Δt_{prot} needed to execute the protocol, i.e. to stop all finished tasks. We cannot, however, compute the complete time Δt_{settle}. Thus we can only safely assume $\Delta t_{settle} \rightarrow \infty$, which from an analysis point of view means, we need to assume the same WCRT for the task in question, even if a higher priority task has stopped to execute.

We make similar observations in the case of merely starting tasks. Consider the jitter diagram of an unchanged task in figure 9. At $t_{on,start}$ tasks are being started and the worst-case response time increases, as more and more tasks of the transition are started. At $t_{on,end}$ all tasks are started in the system and the transition is completed. The WCRT however still increases, as effects of remote tasks might first have to propagate through the system. Only at t_{steady} the system behavior has settled and the WCRT of the unchanged task is equal to $WCRT_{steady,2}$. Also in this case the time t_{steady}, at which all effects of the transition have settled, currently cannot be analyzed by any analysis algorithm. For the scenario of starting additional tasks, this is not relevant though, as $WCRT_{steady,2}$ is always larger that any WCRT during the transition, thus providing a valid upper bound.

Based on these observations, we also think it necessary to redefine the notion of synchrony for Mode-Change Protocols dealing with communicating tasks. The basic property of synchronuous protocols is that finished tasks do not interfere with new tasks. Under the assumption of non-communicating tasks, this is equivalent to the claim that finished and new tasks do not execute at the same time, under the assumption of communicating tasks, this property is harder to establish due to echo-effects. To give credit to these effects, we propose the following new definition of synchronuous Mode-Change Protocols: A Mode-Change Protocol is *synchronuous* if no new task is activated before interference of all finished tasks has ceased to affect *system timing*.

We conclude that, as of today, no Mode-Change Protocol and associated analysis exists that solves the problem of transitioning between configurations in heterogeneuous networked embedded systems with communicating tasks. In the next sections, we will elaborate on scenarios causing a real-time system to transition between configurations, to derive protocols suited to deal with these use-cases. Afterwards, we will evaluate the potential timing issues and analysis difficulties that are likely to be encountered if one wants to utilize such a protocol to finally draw a conclusion on which class of Mode-Change Protocols is safe to use (with respect to timing) in order to build self-organizing real-time systems.

B. Transition scenarios and strategies

We have identified four basic *scenarios* to classify changes in the system configuration:

1) insertion of new tasks
2) removal of already running tasks (finished tasks)
3) combined insertion/removal of tasks (e.g. task migration)
4) change of runtime parameters of an application (e.g. scheduling parameters)

Each scenario comes with its own limitations w.r.t. real-time aspects as we will further discuss in section VIII-C. A configuration change may also be composed of several of these basic scenarios, in which case the most stringent limitations apply.

In terms of managing the transition from one configuration to another, we think it necessary to distinguish between different strategies. As described in the section VIII-A, current mode-change protocols distinguish between synchronous and asynchronous configuration transitions.

Dealing with distributed systems, we propose to extend this classification. We distinguish between protocols that are *globally asynchronous locally asynchronous* (GALA), *globally synchronous locally synchronous* (GSLS) or *globally asynchronous locally synchronous* (GALS). GALA protocols allow parallel execution of new and finished tasks on one processor and on different processors in the system. This strategy is easiest to implement, as tasks of the new configuration may be started at any instance without synchronization on one or between several processors. The opposite case, GSLS, requires that at no time tasks of different configurations may be executed in the system - neither within one processor nor on different processors. Furthermore, according to our extended definition of synchrony, one has to ensure that effects of old-mode tasks have ceased to affect system timing. GSLS protocols can generally only be implemented with additional overhead, as a synchronization between participating processors needs to be established. GALS protocols can be a trade-off. Configuration transitions are performed synchronously on each processor, while different processors perform the change asynchronously. As locally synchronous protocols are comparatively easy to implement and inter-processor synchronization is not required, GALS protocols do not introduce significant overhead.

C. Implications on real-time properties

In this section we will investigate the implications of different combinations of scenarios and strategies on real-time properties of the system. We will present these implications in relation to the required analysis algorithms from section VIII-A. To express these implications, we regard the effects a transition has on the timing behavior of an unchanged task.

For the scenarios of starting and of stopping tasks no finished tasks or no new tasks exist, respectively (unchanged tasks exist in both cases). Thus the three strategies do not differ in their implications on the timing behavior. The corresponding implications have already been discussed in section VIII-A.

For the scenario of combined insertion/removal of tasks, we have to differentiate between the different protocol strategies. From an implementation point of view GALA is the strategy of choice, as synchronization requirements are minimal. Figure 10 depicts the worst-case response time of an unchanged task during a configuration transition with a GALA protocol. At time $t_{sw,start}$ the first configuration change occurs on one resource, starting the transition phase. During Δt_{prot} all resources perform the transition, starting execution of new tasks and stopping execution of finished tasks. During this transition, as GALA is asynchronous, new and finished tasks may exist at the same time. Thus the WCRT may exceed $WCRT_{steady,1}$ and $WCRT_{steady,2}$ - the calculated WCRTs for the two configurations. As discussed before, these transient effects will eventually settle

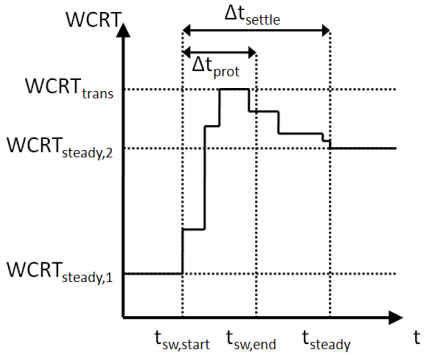


Figure 10: Settling behavior of an unchanged task for the scenario of combined starting and stopping tasks for GALA protocols

at time t_{steady}. However, as t_{steady} cannot be calculated one has to assume the maximum of all three WCRT values $WCRT_{steady,1}$, $WCRT_{trans}$ and $WCRT_{steady,2}$ after the transition as has been done in [HE07], thus leading to accumulated conservatism over several transitions.

We now consider the GSLS strategy. The aim of synchronous protocols is, to avoid the possible load peak, that causes $WCRT_{trans}$. Figure 11 shows the worst-case response time diagramm of unchanged task for a GSLS protocol. At $t_{off,start}$ the transition is initiated. All affected processors start removing finished tasks from the system. At $t_{off,end}$ all finished tasks are stopped. The effects of removing finished tasks settle until t'_{steady}. At $t_{on,start}$ the process of starting new tasks starts and complets at $t_{on,end}$. All transient effects settle until t_{steady}. As can be seen from the WCRT diagram, GSLS protocols successfully avoid the WCRT peak during the transient phase. However, they cannot be implemented. According to the extended definition of "synchronous", the starting of tasks may not be initiated before all transient effects of the finished tasks have settled. Thus $t'_{steady} < t_{on,start}$ is required. As t'_{steady} cannot be determined using state of the art analysis algorithms, the only save assumption is $t'_{steady} \rightarrow \infty$. Thus a save GSLS protocol with current analysis algorithms would require infinite time.

Considering the discussion of the GALA and GSLS strategies, the GALS strategy, which initially seemed like a trade-off between GALA and GSLS, proves to be the least feasible approach. As part of the system is asynchronous, use of GALS strategies may still introduce WCRT peaks on unchanged tasks. At the same time, the GALS strategy requires knowledge about the settling time of the first part of the transition and thus is not implementable.

As indicated in section VIII-A parameter changes can be modelled as a combination of starting and stopping tasks. Thus it imposes the same restrictions on configuration transitions as that scenario.

The results of the discussion on the implications of

strategy	insertion	removal	insertion/removal	parameter changes
GALA	• OK	• unknown settling time	• transient jitter peaks • unknown settling time	• transient jitter peaks • unknown settling time
GSLS	• OK	• unknown settling time	• not implementable	• not implementable
GALS	• OK	• unknown settling time	• transient jitter peaks • unknown settling time • not implementable	• transient jitter peaks • unknown settling time • not implementable

Table II: Overview of implications of transition scenarios and strategies

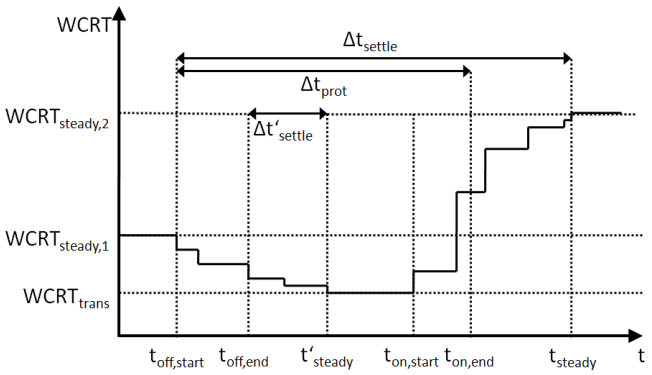


Figure 11: Settling behavior of an unchanged task for the scenario of combined starting and stopping tasks for GSLS protocols

scenarios and transition strategies are summarized in table II.

As can be seen from the table, none of the transition strategies solves the problem of providing consistency between the analyzed model and the actual system behavior. The system designer has options, to enable updates in self-organizing real-time systems though. If, during an update execution of all tasks, including unchanged tasks, is stopped and all communication buffers are reset, thus deleting all state information in the system, every scenario becomes equivalent to the scenario of only starting new tasks. This allows predictable timing-behavior, but makes seemless configuration transitions impossible. Alternatively, the designer has to discard dynamic scheduling and resort to some orthogonalization method (e.g. TDMA). This however usually introduces large overhead.

IX. Conclusion

We have presented a framework and an associated formal protocol ensuring that a self-organizing real-time system can only transition between provenly correct configurations. In this paper, we discussed the issue of consistency in parts arising from the distributed nature of the framework. In order to be able to cooperatively perform the formal protocol in our distributed implementation, we presented a state-synchronization protocol that automatically adapts to reconfigurations (e.g. task migration) in the system.

As a second consistency issue, we have discussed timing implications of mode-changes in distributed real-time systems with communicating tasks. A thorough high-level survey of possible protocols shows that using state of the art analysis algorithms, self-organization in real-time systems can only safely be allowed if dynamic scheduling is discarded or the system state is resetted for each configuration transition.

References

[Bro86] Eugene D. Brooks III. The butterfly barrier. *International Journal of Parallel Programming*, 15(4):295–307, August 1986.

[FHS05] Volker Feil, Cornelia Heinisch, and Martin Simons. Betrachtungen zur Komplexität bei der Aktualisierung von Software im Automobil, 2005.

[GH89] Rajiv Gupta and Charles R. Hill. A scalable implementation of barrier synchronization using an adaptive combining tree. *International Journal of Parallel Programming*, 18(3):161–180, June 1989.

[HE07] Rafik Henia and Rolf Ernst. Scenario aware analysis for complex event models and distributed systems. In *Proceedings Real-Time Systems Symposium*, dec 2007.

[Lee90] C. A. Lee. Barrier synchronization over multistage interconnection networks. In *Proc. Second IEEE Symposium on Parallel and Distributed Processing*, pages 130–133, Dec 1990.

[NGA09] Vincent Nelis, Jol Goossens, and Bjrn Andersson. Two protocols for scheduling multi-mode real-time systems upon identical multiprocessor platforms. *Real-Time Systems, Euromicro Conference on*, 0:151–160, 2009.

[NSSE10] Moritz Neukirchner, Steffen Stein, Harald Schrom, and Rolf Ernst. A software Update Service with Self-Protection Capabilities. In *Proceedings of the conference on Design, Automation and Test in Europe*, 2010.

[RC04] Jorge Real and Alfons Crespo. Mode change protocols for real-time systems: A survey and a new proposal. *Real-Time Syst.*, 26(2):161–197, 2004.

[SHE06] Steffen Stein, Arne Hamann, and Rolf Ernst. Real-time property verification in organic computing systems. *Second International Symposium on Leveraging Applications of Formal Methods, Verification and Validation*, pages 192–197, November 2006.

[TBW94] K. Tindell, A. Burns, and W. Wellings. An extendible approach for analyzing fixed priority hard real-time tasks. *Journal of Real-Time Systems*, 6(2):133–151, 1994.

[YK98] Jenq-Shyan Yang and Chung-Ta King;. Designing tree-based barrier synchronization on 2d mesh networks. *IEEE Trans. on Parallel and Distributed Systems*, 9(6):526–534, June 1998.