# Contract-based Dynamic Task Management for Mixed-Criticality Systems

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

***Abstract*—The use of models is becoming increasingly prominent in the development processes for safety and time critical systems (e.g. in automotive or aerospace). However, oftentimes the models of a component, its implementation properties and execution parameters are only loosely coupled. This missing association complicates system maintainability and becomes an issue with increasing system flexibility.**

**This paper presents a runtime environment closely coupling design-time component models with the execution parameters of the specific component also enabling runtime monitoring of implementation properties. Together with a previously published admission control scheme, this enables tight coupling of component-wise design-time modelling, system analysis and runtime configuration, enabling software flexibility also in mixed-criticality systems.**

## I. Introduction

In many embedded systems, safety and time critical applications share resources with non-critical applications leading to mixed-criticality systems. In such systems components of low criticality must be prevented from affecting the function and timing of critical functions (also required by safety standards, such as ISO 61508). This can be addressed by strict segregation in function and timing (e.g. through TDMA) or by runtime controllers that throttle or stop software components, that exceed their specification.

Reconfiguration of such complex embedded systems is common practice today, e.g. software updates of automotive systems. If no strict segregation between different software components can be assured (e.g. use of a shared CAN bus) complex interdependencies between different software components may exist. As a consequence every update necessitates a verification of the complete system. In addition to the verification, update capabilities require methods to exchange software. In current practice entire binaries are exchanged to update an ECU. In contrast to this practice typical design flows of complex embedded systems follow a component-based approach. Thus, it would be desirable if methods for software replacement also allowed to exchange single software components rather than entire binaries of the affected ECUs. Due to the typical resource limitations (e.g. energy and code size) this has to be done at low overhead. Thirdly, irrespective of the reconfiguration, faults that may occur in single system components, need to be detected and contained to protect other components. This is of particular relevance for mixed-criticality systems, where lower critical applications may be ill-specified but nontheless must not influence highly critical applications. Thus, for reconfiguration of mixed-criticality embedded systems means for *verification*, *reconfiguration* and *isolation* are required.

In current practice these three mechanisms are uncoupled to a large degree. Typically the verification is performed at design-time in the lab. After successful verification, the system is updated in the field. However, this update procedure does not provide the verification model along with the update. Thus, it cannot be ensured that the parameters used for the reconfiguration and isolation mechanisms are consistent with the model used for the design-time verification. Furthermore, it does not allow to check whether the actual implementation of the software component is consistent with the model, that was verified. Thus, if a lower critical application is specified incorrectly the effects on highly critical applications may cause catastrophic failure.

A possible solution is to provide the verification model along with an update and let the embedded system itself check and ensure consistency. In [1] we have already presented a contract-based scheme for admission control that performs a verification of the timing behavior of a system prior to any reconfiguration - thus moving the verification process into the system itself.

In this paper, we aim at coupling such a verification with methods for reconfiguration and fault containment and thus close the gap of current design processes. We show how the verification model, that is used for the admission control, can be closely integrated with the system (re-)configuration and isolation capabilities. We provide a mechanism that configures the system based on the verification model. This coupling ensures that the effects of ill-specified components are properly contained and thus allows safe reconfiguration of mixed-criticality systems. Specifically, we propose a methodology that provides

- dynamic code replacement
- realtime communication
- fault containment in memory and timing

based on the verification model that is used during admission control.

The remainder of this paper is structured as follows. We will first review the admission control scheme of *in-system contracting* as implemented in the EPOC framework (section II). This admission control scheme provides the system

model which we use for task management. In the following we derive key requirements for dynamic task management of mixed-criticality systems and describe the general system architecture (section III). In the three main sections of this paper we will explain the architecture of the communication middleware (section IV), the approach to task management and dynamic linking (section V), and to fault containment (section VI). All these aspects are regarded with respect to data contained in the system model. We will then provide a conclusion (section VII).

## II. Admission Control with In-system Contracting

In this section, we review the principles of in-system contracting and an architecture implementing this admission control scheme.

Current design methodologies, as e.g. specified by AUTOSAR [2], allow the design of software components mainly independent of the underlying hardware platform. Software is partitioned into *tasks* (or *Runnables* in the case of AUTOSAR), i.e. pieces of executable code that can be scheduled on a computational resource, i.e. a processor. Furthermore, communication dependencies (*task links*) between these tasks are specified (similar to *Connectors* across the virtual functional bus of AUTOSAR). In a later design step tasks are mapped to processors and task links to communication resources (i.e. busses or networks). To ensure compliance with realtime requirements, scheduling strategies have to be specified for each resource and timing properties are assigned to each task and task link. A typical characterization of timing is by specification of period and jitter (P,J) in combination with best-case and worst-case execution time (BCET and WCET). With this specification realtime properties can formally be verified using state of the art analysis tools [3], [4].

In the admission control scheme of *in-system contracting* a software application that shall execute on the platform has to provide its verification model as described above. I.e. in the scope of realtime systems this includes the task graph (i.e. tasks and task links), the timing description (e.g. period, jitter, BCET and WCET) and realtime constraints (e.g. path latency). Based on this specification and a model-based verification [5], the admission control mechanism can decide whether all constraints of all software components are satisfied in the presence of that application. If so, the admission control guarantees, that all constraints will be met given that all applications adhere to their specification. The combination of an application specification, its constraints and the guarantees from the admission controller are referred to as a *contract*. Contract-based analysis is an established approach to reduce the verification effort in component-based design [6]. In our case, contract analysis and task management must potentially include the whole system, because of component and task dependenies and because of global constraints such as end-to-end timing. This also requires that all component characteristics are available in the field and are accessible, either directly or by interaction of components.

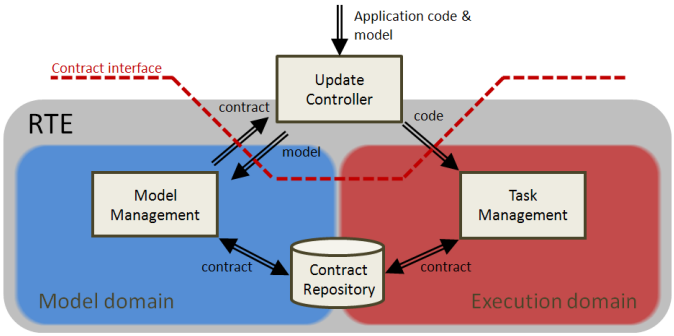



Fig. 1: Framework Architecture

In the following we review the admission control scheme of the EPOC framework [1], which provides the basis for the reconfiguration mechanisms presented in this paper.

### A. EPOC Framework

Fig. 1 shows the EPOC framework architecture. Application updates are inserted into the system via the *Update Controller*. Such an update comprises the actual code and the model of the software application. The model is first inserted into the *Model Management* component within the model domain of the framework [1]. The Model Management component performs a verification based on the supplied model [5] and determines feasibility of the configuration change. If the system determines feasibility of the update, the contract is stored in the *Contract Repository* and returned to the Update Controller. The Update Controller then loads the executable of the software application into the *Task Management* component within the *Execution Domain*. Here the tasks and communication channels are configured according to the model contained in the contract information.

In this paper we focus on the actual reconfiguration process within the Execution Domain based on the data contained in the Contract Repository. Specifically we show how task binaries can be inserted into the system without a dynamic linker, how communication between tasks can be established based on contract data, and how faults occuring in memory accesses and timing can be detected and contained. We will regard these implementation-centric aspects w.r.t. the overall in-system contracting scheme.

### B. Contract Information

In this section we present the data stored within contracts. This data is used to configure tasks and task links, as will be explained in following sections.

Tasks are represented by a structure as shown in table I. Each task is associated with a unique ID (`taskId`) and a timing description as required by the admission control scheme (`actPeriod`, `actJitter`, `wcet`, `bcet`). Furthermore the processor the task is executed on is specified by a unique resource ID (`resId`). Additionally the necessary scheduling parameters are provided (in this case a priority for priority-based scheduling `prio`). Communication dependencies with other tasks are modelled as task links (`*pInLinks`,

```
int         taskId;       // task ID
int         actPeriod;    // activation period
int         actJitter;    // activation jitter
int         bcet;         // best-case execution time
int         wcet;         // worst-case execution time
int         resId;        // resource ID
int         prio;         // scheduling priority
tList       *pOutLinks;   // list of outgoing task links
tList       *pInLinks;    // list of incoming task links
tList       *pServices    // list of used services
tTSDescr    *pTSDescr;    // task slot description pointer
```

TABLE I: Struct for task description

```
int        linkId;      // task link ID
int        srcTaskId;   // source task ID
int        trgTaskId;   // target task ID
int        commId;      // communication resource ID
int        prio;        // communication priority
int        msgSize;     // size of a message
int        msgCount;    // number of messages in buffer
tMsgQ      *pMsgQ;      // message queue
tTDescr    *pTDescr;    // task description pointer
```

TABLE II: Struct for task link description

`*pOutLinks`). Furthermore a task can specify a list of services (`*pServices`) it requires to execute. As last entry a pointer to a descriptor of the *task slot* exists. The specific meaning of the task slot pointer will be explained in section V.

Also task links are managed in a similar data structure (table II). Task links are identified via a unique ID (`linkId`) and the task IDs of source and target task (`srcTaskId`, `trgTaskId`). In case the communication is established via a communication medium, the resource mapping is specified by a communication resource id (`commId`) and scheduling parameters are provided (`prio`). The message size and the number of messages that can be buffered are specified via `msgSize` and `msgCount`, respectively. These parameters determine the best-case and worst-case transmission times and the required message buffer, which we will discuss in section IV. The pointer `*pMsgQ` will point to a message queue that organizes that buffer memory. The description struct for a task link exists twice – once at the source task and once at the target task. `*pTDescr` points to the task description of the task the task link belongs to.

The specification of timing properties, communication dependencies, resource mapping and scheduling parameters is available from the system model, that is used in today's design processes. This data is required for the admission control test. The remaining data can be derived during admission control. It remains static throughout the execution. Thus contract data only changes during system reconfiguration.

In this paper we will show how the execution environment can configure the system based on the above contract information. This ensures that execution parameters are actually set as specified in the system model. Furthermore fault containment and monitoring facilities are configured from contract data. Thus, applications that deviate from their specified behavior are identified and appropriate measures can be taken. This approach guarantees that verification, configuration and implementation are consistent across the system. The presented methodologies are analyzed w.r.t. the overhead they impose.

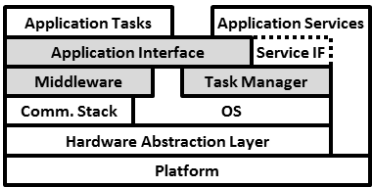



Fig. 2: Architecture Stack

## III. Execution Domain Architecture

In this section, we derive requirements for dynamic task management and introduce our architectural approach. The single components are discussed in more detail in the subsequent sections.

Many existing systems in industrial practice follow an architecture pattern where an operating system and possibly several communication stacks exist on top of a hardware abstraction layer. Application tasks interface directly with the OS as well as the communication stacks. The operating systems support concurrent execution of multiple tasks according to some arbitration scheme and provide basic means of communication and locking, i.e. queues and semaphores, between these tasks. A wide variety of operating systems, that support such features, exist; ranging from sophisticated kernels that support virtualization (e.g. L4 based kernels [7]) down to small-scale operating systems for sensor nodes (e.g. TinyOS [8]). Some OS support dynamic adding and removing of tasks. The communication stacks abstract the underlying communication medium and the associated protocol so that sending and receiving of data chunks is possible. However, message sizes and the addressing scheme typically depend on the communication medium.

Such execution environments do however not provide the necessary mechanisms to couple the configuration of a system with its verification model. As mentioned above, the system specification in e.g. AUTOSAR allows to describe an application largely independent of the underlying platform – the specific configuration of the available communication stacks is then synthesized using this description and a platform specification. Our goal is to diminish the need for this synthesis, by extending the capabilities of the execution environment to be able to provide a service that matches the design abstractions in state-of-the art design processes.

Furthermore reconfiguration should be possible on a per task basis, i.e. reconfiguration of a task does not require the replacement of the entire binary of a processor. Because we consider embedded systems with realtime requirements, the employed mechanisms should impose low overhead on memory and computation time.

To achieve this goal we propose an Execution Domain architecture as shown in fig. 2. As in existing systems an operating system (in our case $\mu$C/OS-II [9]) and a number of communication stacks exist on top of a hardware abstraction layer. The design of these components is not scope of this paper.

The core services of the RTE are implemented on top

of these basic services. A communication middleware as presented in section IV provides resource-agnostic realtime capable communication channels between different tasks. Task links will be synthesized from contract descriptions. The task management component as presented in section V enables dynamic code update and task reconfiguration and provides methods for fault containment. The APIs of the middleware, the operating system, and the task manger are exposed to user applications via the *Application Interface*. Services, i.e. library functions or complex device drivers, may expose an interface to be used by applications, which will be made accessible by the task manager as described in section V.

## IV. Communication Middleware

In this section we describe the reconfigurable communication middleware based on the contracting admission control scheme. Communication between software tasks is also subject to the constraints of embedded realtime systems. Thus, timing properties and resource consumption (i.e. memory, computation and communication time) are critical.

As mentioned above, we assume that distributed applications are decomposed into different communicating tasks. These tasks run in an operating system that abstracts the actual underlying platform and provides an API for e.g. sensor readings. Consequently the implementation of such a task is largely independent of the actual processor it is being executed on, i.e. it is *resource-agnostic*. The communication middleware we are presenting in this section shall provide the same level of abstraction for communication, i.e. abstracting the actual communication medium from the application.

### A. Related Work

Numerous approaches for resource agnostic communication exist in literature. In the domain of distributed realtime and embedded (DRE) systems high-level concepts from general purpose computing have been adopted. Realtime CORBA and DCOM [10], [11], [12], [13] both provide a high level of abstraction by hiding the localization of software objects. An object-request broker (RT-CORBA) or a COM-Server (DCOM) determine the localization of a requested object. A lower layer then determines the route between two objects. Java RMI [14], RCSM [15] and EJB [16] provide comparable services. As localization as well as routing are performed at runtime a high degree of flexibility is offered. Other middleware approaches [17], [18] focus on resource allocation and routing under QoS constraints. All of these services cause a considerable overhead in terms of memory and computation time [19] that may not be tolerable for small scale embedded systems.

Middleware approaches in sensor networks provide communication between nodes at relatively small overhead. Many approaches target data dissemination and collection in the network [20], [21]. Thus they address e.g. dynamic routing [22], data aggregation [23] or rate adaptations to maintain a certain service-quality of the network [24]. Due to the high dynamics that is assumed in these systems, no hard guarantees can be given. Our main concern in the scope of communication middleware however is not highly dynamic runtime adaption but *guaranteed* services in the presence of occasional system reconfigurations.

In the automotive sector a specification for a communication middleware has been developed as one aspect of the AUTOSAR specification [25]. It is particularly designed for safety-critical realtime communication. The underlying software primitives for communication between software modules are synthesized at design-time from a description of software and platform. They support resource-agnostic sender/receiver as well as publisher/subscriber communication. However, no runtime reconfiguration is allowed.

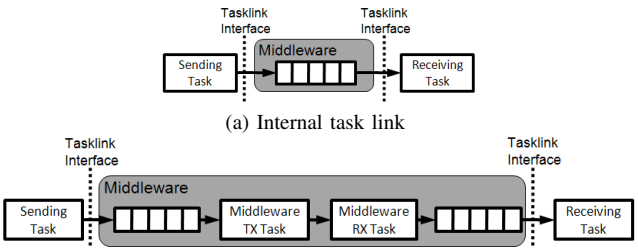
In our in-system contracting scheme reconfiguration is only allowed through admission control to assure hard realtime and safety properties. Thus, the system configuration is only changed at clearly defined points in time. We propose to use contract data for a quasi-static localization and routing scheme. We extract the localization of tasks and the routing of messages from the contracts and configure the middleware accordingly. This approach allows an implementation at minimal overhead while it ensures consistency between contract data and actual execution parameters.

### B. Middleware Implementation

The middleware presented in this paper provides an abstraction similar to the AUTOSAR specification. It allows resource-agnostic sender/receiver as well as publisher/subscriber services. Both are realized through message passing. However, in contrast to AUTOSAR, communication channels are synthesized from contract data during admission control instead of synthesizing them at design-time in the lab. Thus, our middleware approach allows for larger flexibility in the reconfiguration process.

The task management component (see fig. 1) establishes task links during the admission control process. In the case of sender/receiver communication it first determines whether the communication uses a communication resource (external) or is established within one resource (internal). This decides on the implementation of the link (fig. 3). In case of an internal task link a message queue is generated and linked into the task link description (`*pMsgQ`) on sender as well as receiver side (fig. 3a). The message queues are created with the parameters `msgSize` and `msgCount` to enforce adherence to message sizes. For external task links (fig. 3b) a queue is created on sender as well as receiver side and registered at the corresponding task link descriptions. Middleware tasks will handle the communication via the physical communication medium using the transmission parameters from the task link description, i.e. `prio`.

As this setup process is completely executed at admission control time through the task management component based on contract data, the task implementation is guaranteed to be inline with the specification in the model. The entire setup process is invisible to the user application. To use a task link a task first has to query the connection from the middleware




(a) Internal task link

(b) External task link

Fig. 3: Task link implementation (sender/receiver)

providing the task ID of the communication partner. The task ID of the querying task is determined from the operating system to ensure that no communication channels of other tasks can be obtained. The middleware returns the message queue of the task link which is cached in the task. The task can then simply communicate via the message queue irrespective of the location of the communication partner (an example is shown later in alg. 2). Runtime of this communication is bounded, as task link queues are cached within the tasks and the message sizes are upper bounded by the definition in the contract information. The priority-based middleware tasks ensure realtime properties across the communication media, provided the media themselves are realtime capable.

Publisher/subscriber connections are established in a similar fashion as sender/receiver connections. This is possible, because all subscribers are known at admission control time. In case of publisher/subscriber communication each subscriber maintains a separate message queue. As posting to a publisher/subscriber task link may requires sequential copying of the message to all receiver queues, the runtime depends on the number of subscribers. To bound the runtime the number of possible subscribers has to be limited to a fixed value.

The presented middleware architecture has some key advantages. First, localization and routing are defined through contracting and links are established at admission control time, thus reducing possible execution-time overhead. Secondly, tasks can only query their own task links, providing a certain degree of isolation. Finally, sender/receiver as well as publisher/subscriber communication are realized via the same API, providing an easy to use interface.

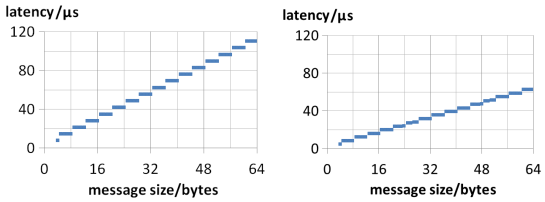
In contrast to a lab-based approach, as in AUTOSAR, reconfiguration of communication channels can be performed in the system without the necessity to re-synthesize communication links in the lab and to replace the corresponding application binaries.

### C. Overhead Evaluation

In this section we present the overhead the communication middleware imposes on memory consumption and timing properties. For evaluation purposes we are using two different hardware platforms: a physical setup consisting of two MPC5200 evaluation boards interconnected by CAN bus and a simulated hardware platform consisting of two ARM926E processors at 100MHz communicating over a CAN bus. The simulation is performed using the cycle-accurate system simulator CoMET [26]. The code was written in plain C.

| Module | ROM (kB) | RAM (kB) |
|---|---|---|
| **Communication middleware** | | |
| Generic functions | 2,6 | 0,1 |
| CAN specific | 2 | 0,5 |
| **Task management** | | |
| Generic functions | 4,9 | 1,7 |
| Jumptable | 0,1 | 0 |
| Migration specific | 1 | 0 |
| **Memory Protection** | | |
| Total | 1,8 | 0.5 |

TABLE III: Memory Consumption




(a) Sending

(b) Receiving

Fig. 4: Latencies of internal task links

For evaluation of the communication middleware we have used the simulated platform, as it allows easy measurement of system timing. The memory overhead the middleware imposes is shown in table III. The program code requires 4,6kB of memory while 0,6kB are required for data. About 2,6kB of the program code is required for generic functions, such as sending and receiving to/from task links. The remaining 2kB are used for functions specific to the CAN bus. These include e.g. the initialization of the CAN controller and the middleware TX and RX tasks. The largest portion of the required RAM is used for the stacks of these middleware tasks.

To determine the overhead on latency the middleware imposes we have made measurements for different message sizes (several bytes per message). Figure 4 shows the delays for sending and receiving on internal task links for different message sizes. The delays are in the order of magnitude of $10\mu$s while they scale linearly with message size. The latency imposed through the TX and RX tasks are dominated by the significantly slower CAN bus. The total delay of the tasks and the communication medium lie in the order of magnitude of 1ms.

## V. Task Management

In addition to a communication middleware, the framework supports dynamic task management. This is mandatory, if parts of the system shall be exchanged at runtime, e.g. during an update. In this section, we first highlight the key challenges to be tackled when implementing dynamic task execution in an embedded system, to then introduce the specific solutions implemented in the EPOC framework.

### A. Challenges

The first challenge to be tackled is the establishment of position independence of the code to be dynamically executed.

As embedded systems may not support virtual memory due to resource constraints, tasks must be able to run at any memory location on the target platform. A major problem is the interaction with the operating system infrastructure on the target platform. This issue will be discussed in section V-B.

Once position independent code is available, the question of managing the task's binary as well as its associated static and dynamic data must be answered. Our approach to organizing tasks in memory is described in section V-C. Once this is established, we introduce the complete flow of starting applications in the EPOC framework.

Once the dynamic management of tasks within the EPOC framework is fully described, we outline the programming paradigms a designer has to follow when implementing an application for this platform to show that the design decisions taken do not overly complicate the task of writing a piece of software on such a framework (section V-E).

### B. Position Independence of Tasks

In order to support updates where replacement of the full binary is not an option, dynamic task or code management must be enabled. In desktop operating systems, this is usually achieved by dynamic linking and virtual memory partitions as can be provided by memory management units of modern processors. This setup does however result in a considerable overhead both in operating system logic (dynamic linking) and hardware requirements (MMU), which is often not tolerable.

A second approach, originally used in the context of server-side computing, but also extended to embedded systems and small-scale devices is the use of a virtual machine to host the code. This approach has been followed by the Maté VM as presented in [27]. Although this approach yields a high level of abstraction of the target code from the actual runtime environment and allows a compact representation of programs, it requires the application developer to learn the specific programming language of the VM. This does not integrate easily into the design flow of modern embedded systems and forbids the use of legacy code.

Using standard C as a programming language, code segments can however be compiled and linked independently of their actual memory in the code, by only referencing data and performing jumps relative to the program counter or a given offset register. This is a common feature in C compiler suites, such as gcc. The runtime environment API can be made available by means of a *jumptable*, that provides access to the runtime environment API at defined addresses in memory. This approach has been presented in [28], where it has also been shown that this approach yields high flexibility while keeping the necessary overhead at a minimum. [29] showed that the jumptable approach can be replaced by runtime dynamic linking, without requiring a MMU. The implementation of the dynamic linker presented in [29] comprises of roughly 6kB of code on a 16 bit machine.

To avoid this overhead we follow the jumptable approach to dynamic code management as presented in [28]. This approach achieves absolute jumps to API functions by a custom set of assembly functions that incur absolute jumps to given memory addresses. Alg. 1 shows an example of a function from the application interface. When this API function is called, the function arguments are loaded into the registers and an absolute jump to the appropriate RTE function is performed. As the assembly is designed as inline assembly functions, their use looks like a normal function call in the application code. This enables the application developer to design the code of his application without the need to link against a runtime environment binary or the actual code. Thus, for the RTE designer, it is sufficient to distribute the assembly macro set to potential designers of applications, enabling a distributed software development process with strict separation of concerns between operating system/RTE and application developers. Integration of services, i.e. library functions such as e.g. complex device drivers, is also designed as described in SOS [28]. Software modules can register a service API with the task manager, which exposes it via the application interface. The API is represented by a set of function pointers at given offsets from the start of the service binary. This enables an easy and modular extension of the runtime environment. Note however, that due to the multithreaded base operating system, services must in general be *reentrant*, i.e. it must be possible to start a novel invocation of an API function before the prior one has finished. In case this is not possible, as the service e.g. represents a shared hardware on the processor, the service call has to be protected by a semaphore and the resulting blocking time has to be considered by the admission control scheme.

**Algorithm 1** Example of an assembly macro for jumptables

```
1:  asm void ApplicationInterfaceFunction(int arg1, int arg2){
2:  // copy arguments into appropriate registers
3:  mr r3,arg1
4:  mr r4,arg2

5:  // branch and link to the appropriate function
6:  lis r11,ApplicationInterfaceCallFunction@ha
7:  addi r11,r11,ApplicationInterfaceCallFunction@l
8:  mtspr ctr,r11
9:  bctrl
10: }
```

### C. Memory Organization

Position independet code allows large freedom in memory organization as code can be freely relocated. In this section we present the memory layout of the EPOC framework, which is optimized to reduce the overhead in task management and memory protection.

In the EPOC framework, tasks are managed in *task slots* that contain not only the binary image of the task, but also all relevant memory structures needed by that task. Allocating all memory of user tasks in a contiguous task slot enables efficient use of a memory protection unit, as will be later shown in section VI-A.

The memory layout of a task slot is depicted in fig. 5. The first area of the task slot is dedicated to the task's stack. The location of the stack area is motivated by the memory protec-

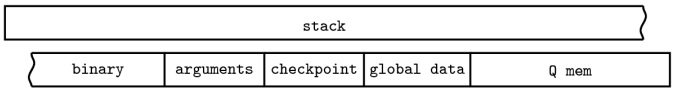



Fig. 5: Task Slot Memory Layout

```
void      *pStart;          // start of task slot pointer
int       taskOffset;       // task function offset
int       taskArgOffset;    // task arguments offset
int       migOffset;        // migration memory offset
int       migLength;        // migration memory length
int       dataOffset;       // data memory offset
int       dataLength;       // data memory length
int       commMemOffset;    // comm. queue memory offset
int       commMemLength;    // comm. queue memory length
tTDescr   *pTDescr;         // task description pointer
tTState   state;            // current state of task
tFlag     *pFlag;           // signalling flag pointer
```

TABLE IV: Struct for task slot description

tion scheme (see section VI-A). Then, the binary, followed by the data to be passed to the task is allocated. After this, in case the task shall be able to start from a previously saved state, a memory area for construction of checkpoints is allocated. This area is followed by memory space for global variables of the task. The upper memory region of a task slot contains memory for the middleware queues needed by this task. Note that these are manged by the middleware as described before. In case of internal task links, the memory of the queue is allocated in the sender's task slot.

Depending on memory availability the number of available task slots may differ between different processing nodes of the platform. Also the allocation in memory and the size of a given task slot may vary. In order to manage tasks within their slots, the task manager maintains an array of task information structures as outlined in table IV. It contains a pointer to the start of the given task slot (`*pStart`), offsets from this base pointer and sizes of the relevant memory regions within the task slot (`...Offset-...Length` pairs). In case of equally sized task slots, this enables a quick update of a task description in case a task is moved to another slot, e.g. in case of task migration. Only the base pointer needs to be updated.

Additionally, this allows efficient introduction of tasks into the system from an outside source. As the offsets can be calculated offline after compilation, a descriptive struct as above can be provided together with the binary. Once this data is copied into the processors' memory, only the base-pointer needs to be updated to reflect the correct memory location.

In addition to these pointers, the information structure also contains a pointer to the task's contract data (`*pTDescr`) as well as information about the state of the task and a pointer to a signalling flag for communication with the running task (`state` and `*pFlag`). A pointer to this runtime description of the tasks is also kept within the contract of the specific task.

### D. Task Start-up

Until now we have shown how tasks are organized and how communication among them is established. In this section we describe the steps taken to actually configure and start a new task. These steps are performed by the Task Management component based on contract data that has been verified by the admission control scheme. This ensures that the actual execution parameters are consistent with the contract.

We assume that the tasks binary (and potentially all parameters the task requires for startup) together with a descriptive struct as shown in table IV have been placed in memory after admission control. The Task Manager is passed a pointer to the descriptive struct in order to insert the given task into a task slot and commence its execution.

In order to start the task, the provided data is copied into the task slot that has been allocated by the admission control scheme. In the descriptive struct only the pointer to the start of the task slot (`*pStart`) needs to be updated accordingly. All offsets and lengths remain unchanged. Then the associated contract data is used to initialize the middleware queues required for the task's task links. The backing memory of outgoing task links is allocated within the task slot from memory starting at `*pStart+commMemOffset`.

In case the task is time-triggered (i.e. an activation period is specified in the contract data), a timer is configured to provide an activation service to the task. Note that multiple tasks may share a single timer. Also a signalling flag is allocated to provide communication between the running task and the RTE.

Once this is done, the startup parameters are initialized with pointers to checkpoint and global memory as well as the signalling flag. Now that configuration is complete the task can be started using the appropriate service function of the underlying operating system. The necessary scheduling parameters are read from contract data to guarantee correctness.

From this startup sequence we see that the entire configuration process is controlled through the task management component. All parameters are set from data from admission control. The application task does not even require access to this configuration data. This process ensures that the configuration is consistent with the model that was verified during admission control.

### E. Application Interface

In order to correctly interact with the presented framework, an application designer has to adhere to a specific design pattern. Alg. 2 shows a code example of a typical task. During an initialization phase, the pointers to the migration and global memory space should be extracted and cached from the memory area passed as an argument (lines 6 to 8). As shown in the example, the programmer may decide to impose a predefined structure over these memory regions. As a second step, pointers to relevant task links should be queried and cached to avoid query overhead during runtime of the task (lines 9 and 10). Next, an optional section to initialize the task from a previously saved checkpoint can be implemented, in case this task should be migratable without losing state information (line 12). Once this initialization work has been performed, the task may enter its main loop (line 16) that contains the actual functionality of the task.

In case the task is event-triggered message reception must be performed as first action (line 18). The timing model of

**Algorithm 2** Code example of a task

```
 1: void Task(void *voidTaskArgStruct){
 2: tMsg msg;
 3: // Cast argument struct
 4: tArg *pArg = (tArg*)voidTaskArgStruct;

 5: // Get and cache relevant pointers
 6: tFlag *pSignallingFlag = pArg→pSignallingFlag;
 7: tMigrationMem *pMigrationMem = pArg→pMigrationMem;
 8: tGlobalMem *pGlobalMem = pArg→pGlobalMem;
 9: tMsgQ *pInLink = TM_GetTaskLink(1);
10: tMsgQ *pOutLInk = TM_GetTaskLink(2);

11: // initialize from checkpoint
12: if pArg→checkpoint == true then
13:     // initialize from pMigrationMem
14: end if

15: // start main loop
16: while (true) do
17:     // Activation through message reception or timed activation
18:     MW_ReceiveFromTaskLink(pInLink,&msg); // activation by message
19:     // TM_WaitForTimedActivation(); // activation by timer

20:     // Regular processing here

21:     if *pSignallingFlag == true then
22:         // create checkpoint in pMigrationMem
23:         *pSignallingFlag = false;
24:     end if
25:     // Send Message to subsequent task
26:     MW_SendToTaskLink(pOutLink,&msg)
27: end while
```

the admission control scheme assumes that messages are sent at the end of the execution. This is not mandatory – see [30] for a thorough discussion.

In case the task supports migration and may restart from a checkpoint, the main loop must contain logic to store a checkpoint in the migration memory area. This code shall be executed, as soon as the RTE triggers the signalling flag (line 21). In order to enable efficient encoding of checkpoints, the application designer should query the status of the flag at convenient places in the code and - if set - construct a checkpoint and clear the flag to complete the handshake.

### F. Overhead Evaluation

As task management is only active during occasional transition phases and can be performed as a background task, its timing is not critical for system functionality. Thus, we only consider its overhead in means of memory consumption as summarized in table III. Our implementation entails a total of 7.7kB memory overhead on our 32 bit ARM evaluation platform, most of which (4.9kB) is necessary logic. The remaining 2.8kB distribute over necessary data structures (i.e. task slot descriptions), the jumptable and migration specific code for handling checkpointing and code transfer. Thus, although providing a great amount of flexibility in the design and evolution of mixed-criticality systems, it does not add a large overhead in terms of memory to the overall system. Note that adding a dynamic linker [29] would add a considerable amount of additional overhead (6kB on a 16 bit machine as in the implementation of [29]).

## VI. Fault Containment Facilites

By now, we have described our approach to reconfiguration and dynamic management of tasks in a low-overhead environment. Together with the admission control scheme [1], such an environment can ensure that every configuration change is verified and properly realized. However, implementation of software modules may be faulty and not adhere to their description - especially for low criticality applications where requirements on qualification may be low. Thus, running tasks must be supervised w.r.t. their contracts and occuring faults must be contained, such that components of higher criticality are not affected.

In this paper we consider two fault scenarios - memory access violations and violation of timing specifications. Both may easily occur due to e.g. programming errors and both are hard to detect and debug with existing design processes. In this section we show how the presented approaches to task management and communication can be exploited to accomplish memory protection and containment of timing errors in the embedded system at very low overhead. Also these containment mechanisms are configured from contract data to ensure consistency.

### A. Memory Protection

To provide the same level of abstraction as current design processes, memory protection shall be established on a per task basis. We aim to provide protection between different application tasks as well as protection of the runtime environment. As we consider fault containment (not security) only write accesses and execution rights are critical.

Memory protection can be achieved either by hardware through use of a memory management unit (MMU) or a memory protection unit (MPU) or in software as presented in e.g. [31], [32]. Most software approaches are based on rewriting compiled binaries. Instructions that are writing to memory are replaced by an instruction sequence that checks access rights first. Especially for data intensive programs this causes tremendous overhead.

In order to keep runtime overhead low, we consider memory protection by hardware support rather than software solutions. The approach we present in this section exploits the memory layout of the task slots and effectively confines the effects of programming errors to memory areas used by the erroneous task. The memory layout was designed to take advantage of a memory protection unit if present in the given system. We shortly introduce the capabilites of the MMU available on the MPC5200 evaluation board to then describe the memory protection scheme we superimposed on the memory layout in our implementation. To assess the effectiveness and overhead of this solution, we discuss which kind of memory access errors can be detected by the solution as well as the memory and computation overhead incurred by it.

The Freescale MPC5200 microcontroller implements a PPCe300 core which contains an MMU providing two schemes for memory protection, namely Block Address Translation (BAT) and Page Address Translation (paging). Both

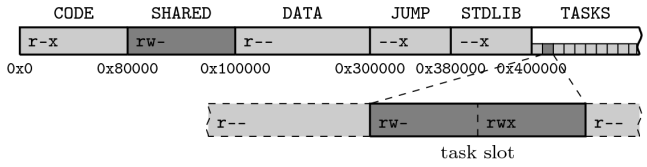



Fig. 6: Memory Alignment for Memory Protection

| Code Role | # of instructions |
|---|---|
| OS context Switch | 38 |
| Invalidate TLB and BAT Registers | 163 |
| Initialize TLB and BAT for own task slot and RTE | 106 |
| Initialize TLB for access to other task slots (per slot) | 97 |

TABLE V: Memory Protection context switch overhead

schemes support not only memory protection, but also address translation, which we do not take into account in the scope of this paper as it is an uncommon feature in e.g. automotive systems. Both approaches enable the user to assign access permissions (read, write, execute) to memory regions, to raise an interrupt in case a non-configured or non-allowed access to a memory region is performed.

The two schemes differ in the size and number of individual memory regions, they can supervise. On the MPC5200, BAT supervises up to eight memory regions of at least 128 kB up to 256 MB. For paging the MMU implements a Translation Lookaside Buffer (TLB) to store up to 64 entries covering 4 kB each. As a consequence, we use BAT to protect the large code chunk representing the RTE and paging to configure the individual task slots, as well as shared memory areas. For this reason, we align the memory areas as shown in fig. 6 with possible boundaries and sizes for BAT and paging. Task slots are split into two memory regions, one that holds the stack and another one that stores the actual instructions and data. In this example, each of these regions is exactly one page in size. Of course, they may be scaled in multiples of this size, if larger tasks need to be supported. The figure also shows the access rights of an application task during execution.

As can be seen from fig. 6, each task of an application only has write-access to a limited amount of the memory. Thus, the memory protection scheme efficiently protects correct tasks from faulty ones, as these cannot corrupt their memory. Allocating the stack at the beginning of a task slot has the advantage that stack overflows, otherwise hard to find, can be detected by the memory protection scheme. As stacks on our platform grow downwards and adjacent task slots are not writable by the given task, a growth of the stack over the stack boundary immediately causes a memory access violation.

Note that the memory address 0x0 is not writable in the task context. If memory is zeroed at startup many illegal accesses due to uninitialized pointers can be identified.

In addition to restrictions on write accesses, execution rights are restricted for all tasks. Only the task's own program code section as well as the sections containing program code of the RTE are executable, thus providing additional protection.

All memory access rights are configured by the Task Manager from data from the task slot description (table IV). As this data is verified or generated through admission control consistency with the verification is ensured.

The memory protection as outlined in this section causes only low overhead on memory. In our implementation, it added 1.8 kB of program code and 0.5 kB of data to the RTE binary. It does however also introduce computational overhead, which we discuss in more detail in the following. As the memory access layout is potentially different for each task running on the system, reconfiguration of the MMU/MPU is required during each context switch. The additional overhead during the context switch due to reconfiguration of the MMU is summarized in table V. Without memory protection enabled, a context switch takes 38 instructions to complete. The two necessary steps of invalidating the old configuration and initializing the access pattern for the task to be scheduled takes 163 and 106 instructions, respectively. In case the upcoming task has incoming internal task links, the corresponding message queues in the senders' task slots must be readable. Thus, for each internal incoming task link, an additional TLB entry must be configured, which infers an overhead of 97 instructions. The total context switch overhead can be bounded by bounding the number of admissable internal incoming task links per task.

### *B. Timing Fault Containment*

The presented framework not only allows to detect and prevent memory access violations effectively and efficiently, but also allows to detect and confine timing errors. In this section we show how the timing fault containment is achieved through configuration and monitoring based on contract data.

To ensure temporal isolation execution times and activation patterns have to be monitored/controlled by the RTE [33]. As described in section II-B contract information provides an appropriate specification with activation periods, jitter and best-case and worst-case execution times. If the implementation of a software task is not consistent with this specification, e.g. it executes longer than specified or it is activated too frequently, this may influence the timing behavior of other software modules in an invalid way, e.g. through longer preemption in priority-based preemptive scheduling. This may lead to timing behavior that was not verified through admission control and that thus may cause illegal interference between different software components.

The framework presented in this paper allows to easily check or enforce adherence of software modules to their own temporal specification. Activation of time-triggered tasks is readily enforced through the task management component. As shown in the code example of a task (algorithm 2) a task has to request its next activation from the RTE. The RTE reads the activation period from the contract information and triggers the activation accordingly. The task has no means to circumvent this process and thus cannot violate its requirement on activation period. Similarly, the transmission time on comm. resources is directly enforced through the communication middleware. As described in section IV task links are set up with communication buffers according to the contract de-

scription (`msgSize, msgCount`). When sending data along a task link, the middleware only transmits `msgSize` bytes from these buffers. This ensures that worst-case transmission times of inter-task communication are not exceeded. Thus, time-triggered activations and message transmission times are directly enforced through configuration from contract data.

Metrics that remain to be monitored are (i) execution times of tasks, (ii) the activation period and jitter of tasks, that are activated through some external event and (iii) period and jitter of message transmissions. These metrics can easily be monitored by means of watchdog timers [34], timed monitoring tasks [35] or heartbeat monitors [33]. The required configuration parameters can be drawn from contract data. In case the monitoring facility detects a violation of the specification the violating task can be stopped from executing.

## VII. Conclusion

In this paper we have presented a methodology for contract-based dynamic task management for mixed-criticality realtime systems. The presented approach provides a close coupling between an admission control scheme and task management facilities to ensure consistency between the formally verified model and the configuration.

We have shown how the verification model that is used during admission control can be used for system (re-)configuration. The employed reconfiguration mechanism allows exchange of software binaries of single tasks and provides means for resource-agnostic communication between these tasks. Furthermore we have shown how the dynamic task management approach allows containment of memory access and timing faults also based on contract data. Thus the presented methodology integrates the steps of verification, configuration and fault containment by the means of contracting. This allows software reconfiguration also for hard realtime and safety critical mixed-criticality systems.

## References


[1] 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.

[2] N. Tracey, U. Lefarth, H.-J. Wolff, and U. Freund, "Ecu software module development process changes in autosar," ETAS GmbH, Borsigstr. 14 70469 Stuttgart, Tech. Rep., 2007.

[3] 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.

[4] L. Thiele, S. Chakraborty, and M. Naedele, "Real-time calculus for scheduling hard real-time systems," in *Proc. of Int'l. Symp. on Circuits and Systems (ISCAS)*, 2000.

[5] 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.

[6] A. Benveniste, B. Caillaud, A. Ferrari, L. Mangeruca, R. Passerone, and C. Sofronis, "Multiple viewpoint contract-based specification and design," in *Formal Methods for Components and Objects*. Springer, 2008, pp. 200–225.

[7] G. H. Alan Au, *L4 User Manual*, School of Computer Science and Engineering, The University of New South Wales, Sydney 2052, Australia, March 1999.

[8] P. Levis and D. Gay, *TinyOS Programming*. Cambridge University Press, 2009.

[9] J. J. Labrosse, *MicroC/OS-II - The Real-Time Kernel*, 2nd ed., 2002.

[10] D. C. Schmidt, "Middleware for real-time and embedded systems," *Commun. ACM*, vol. 45, no. 6, pp. 43–48, 2002.

[11] D. C. Schmidt and C. O'Ryan, "Patterns and performance of distributed real-time and embedded publisher/subscriber architectures," *Journal of Systems and Software*, vol. 66, pp. 213 – 223, 2003.

[12] C. D. Gill, D. L. Levine, and D. C. Schmidt, "The design and performance of a real-time corba scheduling service," *Real-Time Systems*, vol. 20, pp. 117–154, 2001.

[13] P. E. Chung, Y. Huang, S. Yajnik, D. Liang, J. C. Shih, C. Wang, and Y. Wang, "Dcom and corba side by side, step by step, and layer by layer," 1997.

[14] A. Wollrath, R. Riggs, and J. Waldo, "A distributed object model for the javatm system," in *Proc. of the 2nd USENIX Conf. on Object-Oriented Technologies (COOTS)*, 1996.

[15] S. Yau, F. Karim, Y. Wang, B. Wang, and S. Gupta, "Reconfigurable context-sensitive middleware for pervasive computing," *Pervasive Computing, IEEE*, vol. 1, pp. 33 – 40, 2002.

[16] A. Thomas, "Enterprise javabeans technology," Sun Microsystems, Inc., Tech. Rep., 1998.

[17] T. Abdelzaher, S. Dawson, W.-C. Feng, F. Jahanian, S. Johnson, A. Mehra, T. Mitton, A. Shaikh, K. Shin, Z. Wang, H. Zou, M. Bjorkland, and P. Marron, "Armada middleware and communication services," *Real-Time Syst.*, vol. 16, pp. 127–153, 1999.

[18] T. Cucinotta and L. Palopoli, "QoS Control for Pipelines of Tasks using Multiple Resources," *IEEE Trans. on Computers*, vol. 59, pp. 416–430, 2010.

[19] D. C. Schmidt, M. Deshpande, and C. O'Ryan, "Operating system performance in support of real-time middleware," in *In Proc. of the 7th IEEE Workshop on*, 2002.

[20] E. Souto, G. Guimar aes, G. Vasconcelos, M. Vieira, N. Rosa, and C. Ferraz, "A message-oriented middleware for sensor networks," in *MPAC '04: Proc. of the 2nd workshop on Middleware for pervasive and ad-hoc computing*, 2004.

[21] P. Levis, N. Patel, D. Culler, and S. Shenker, "Trickle: a self-regulating algorithm for code propagation and maintenance in wireless sensor networks," in *NSDI'04: Proc. of the 1st Symp. on Networked Systems Design and Implementation*, 2004.

[22] J. N. Al-Karaki and A. E. Kamal, "Routing techniques in wireless sensor networks: a survey," *IEEE Wireless Communications*, vol. 11, pp. 6–28, 2004.

[23] L. Krishnamachari, D. Estrin, and S. Wicker, "The impact of data aggregation in wireless sensor networks," in *Distributed Computing Systems Workshops, 2002. Proc.. 22nd Int'l. Conf. on*, 2002.

[24] X. Yu, K. Niyogi, S. Mehrotra, and N. Venkatasubramanian, "Adaptive middleware for distributed sensor environments," *IEEE Distributed Systems Online*, vol. 4, 2003.

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

[26] "Synopsys, inc., http://www.synopsys.com," Internet.

[27] P. Levis and D. Culler, "Maté: a tiny virtual machine for sensor networks," *SIGOPS Oper. Syst. Rev.*, vol. 36, pp. 85–95, 2002.

[28] C.-C. Han, R. Kumar, R. Shea, E. Kohler, and M. Srivastava, "A dynamic operating system for sensor nodes," in *MobiSys '05: Proc. of the 3rd int'l. conf. on Mobile systems, applications, and services*, 2005.

[29] A. Dunkels, N. Finne, J. Eriksson, and T. Voigt, "Run-time dynamic linking for reprogramming wireless sensor networks," in *SenSys '06: Proc. of the 4th int'l. conf. on Embedded networked sensor systems*, 2006.

[30] K. Richter, "Compositional scheduling analysis using standard event models," Ph.D. dissertation, Technical University of Braunschweig, Department of Electrical Engineering and Information Technology, 2004.

[31] R. Kumar, E. Kohler, and M. Srivastava, "Harbor: Software-based memory protection for sensor nodes," in *Information Processing in Sensor Networks, 2007. IPSN 2007. 6th International Symposium on*, 2007.

[32] L. Gu and J. A. Stankovic, "t-kernel: providing reliable os support to wireless sensor networks," in *Proc. of the 4th int'l. conf. on Embedded networked sensor systems*, 2006.

[33] X. Chen, J. Feng, M. Hiller, and V. Lauer, "Application of software watchdog as a dependability software service for automotive safety relevant systems," in *Dependable Systems and Networks, 2007. DSN '07. 37th Annual IEEE/IFIP Int'l. Conf. on*, 2007.

[34] K. Tindell, H. Kopetz, F. Wolf, and R. Ernst, "Safe automotive software development," in *Design, Automation and Test in Europe Conference and Exhibition*, 2003.

[35] S. Chodrow, F. Jahanian, and M. Donner, "Run-time monitoring of real-time systems," in *Proc. of Real-Time Systems Symp. (RTSS)*, 1991.