FAKULTÄT FÜR !NFORMATIK

Validation of the DECOS Encapsulated Execution Environment

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Informatik

eingereicht von

Thomas Strnad
Matrikelnummer 9550989

an der
Fakultät für Informatik der Technischen Universität Wien

Betreuung:
Betreuer: o.Univ.Prof. Dr.phil. Hermann Kopetz
Mitwirkung: Univ.Ass. Dipl.-Ing. Harald Paulitsch und
Univ.Ass. Dipl.-Ing. Dr.techn. Astrit Ademaj

Wien, 30.10.2008

(Unterschrift Verfasser) (Unterschrift Betreuer)

FAKULTÄT FÜR !NFORMATIK

Validation of the DECOS Encapsulated Execution Environment

DIPLOMA THESIS

for obtaining the academic degree

Diplom-Ingenieur

in the course of the studies

Informatik

filed by

Thomas Strnad
Matr. number 9550989

at the
Fakultät für Informatik der Technischen Universität Wien

Supervision:
Supervisor: o.Univ.Prof. Dr.phil. Hermann Kopetz
Advisors: Univ.Ass. Dipl.-Ing. Harald Paulitsch and
Univ.Ass. Dipl.-Ing. Dr.techn. Astrit Ademaj

Vienna, 30.10.2008

Abstract

The DECOS (Dependable Embedded COmponents and Systems) integrated architecture allows the integration of different embedded application subsystems with different criticality into the same hardware infrastructure. In the DECOS integrated architecture, computational resources (CPU time, memory, I/O) and communication resources (network bandwidth) are shared among multiple software components in order to reduce the number of deployed embedded computer nodes, which implies the reduction of system cost.

The DECOS integrated architecture consists of four layers: the I/O layer, the application layer, the middleware layer, and the core layer. Distributed software applications run at the application layer. So called DECOS high-level services (virtual network service, virtual gateway service, diagnostic service) are executed in the middleware layer. The core layer provides services that are in charge of predictable and fault-tolerant communication among different DECOS integrated nodes.

In order to perform a seamless integration of different software modules that may be developed by different vendors, the DECOS integrated architecture services guarantee that different applications do not affect the operation of each other in an undesired manner: An application job that is executed in one of the DECOS components can not affect other application jobs or DECOS services. A prototype implementation of the DECOS integrated architecture was developed at the Vienna University of Technology.

The encapsulated execution environment is in charge of preventing non-specified interaction among the application jobs (implemented as LXRT tasks) executed in the DECOS components. The encapsulated execution environment is implemented by using the Linux operating system with RTAI and LXRT patches. The objective of this work is to validate whether Linux-RTAI-LXRT fulfills the requirements to be used as an encapsulated execution environment in the DECOS integrated architecture. Validation is performed by means of software implemented fault injection (SWIFI).

SWIFI is usually deployed to emulate the occurrence of hardware faults. In this work, SWIFI is used to perform the emulation of software faults in order to observe, if a faulty application job that is executed in a DECOS component can affect the operation of other application jobs or the DECOS services.

Acknowledgements

This thesis was partly supported from the European IST-FP6 Integrated Project DECOS. I want to thank the head of the Real-Time System Group at the Technical University of Vienna, Prof. Hermann Kopetz, for the opportunity to contribute and develop my thesis within this project.

My special thanks go to my supervisor Astrit Ademaj for his great support throughout the work, even after he left for TTTech. He gave me invaluable advice in the field of software fault injection, sourcing scientific papers, and the scientific writing. Without his help regarding the DECOS concept and reviews of implementation details, this work would not appear in its present form.

Furthermore I want to thank Harald Paulitsch for his comments during proof reading and his great advice regarding reader-friendly scientific papers.

I would like to acknowledge Peter Hammer, my supervisor at work at Cincinnati Extrusion, who enabled me to complete this thesis even during my full-time employment.

I also would like to thank my friends for their appreciation when I was busy spending time on the thesis.

Thanks also to my parents Renate and Karl and my sister Verena for supporting me throughout the years. To them I dedicate this thesis.

Contents

1 Introduction

Embedded systems are the core components in many automated systems. They perform the control of operations in devices ranging from household washing machines up to safety critical applications like flight coordination of aircrafts. With the integration of more and more functions in such systems, multiple networks were connected to distributed systems in order to segment the workload.

Current applications are often realized by using components of different characteristics and connect them by a communication system, e.g. a bus system like the CAN bus, to distribute information among the various components. This design principle is called a *federated* system design. Each functionality in a system requires a separate embedded computer node. Each node executes a software task, and a set of all tasks in such an embedded computer network is denoted as a *Distributed Application System (DAS)*.
The design which allows to reduce the number of DAS in a distributed system and integrate several independent DASes in a single computer system is called an *Integrated architecture*.
The move from a federated to an integrated architecture requires the proof that applications integrated into a single node do not influence each other under all conditions. One must be able to guarantee the logically and timely correctness of the whole system by reproducible test cases. For the certification process of safety critical systems, the validation of these claims is a must.

One has to use a dedicated framework to provoke the several fault cases stated in the design specification of integrated systems under the term *fault model*. The framework to inject faults into the system shall introduce an overhead as small as possible to not influence the default system behavior unintentionally (the probe effect). Fault injection tools are required, because faults of the fault hypotheses normally appear only in rare cases. Injection by synthetic tools allows the accelerated occurrence of defined faults in a computer system.

This thesis deals with the several already existing software implemented fault injection (SWIFI) tools for real-time systems and introduces a customized SWIFI framework for the real-time system under test.

Eventually, test cases for the fault model of this specific system are presented and validated. The results derived from this validation pass can give hints for further improvements of the current implementation.

1.1 Problem Definition

DECOS integrated architecture allows integration of different embedded application subsystems with different criticality into the same hardware infrastructure. The encapsulated execution environment (EEE) and the DECOS middleware services and core services are in charge of preventing non specified interaction among the application jobs and prohibit monopolization of the communication medium by a faulty task or component. One possible design is to implement the encapsulated execution environment by using the encapsulation mechanism (allocation of memory and CPU resources to specific tasks) of operating systems. The Linux operating system was used in a prototype implementation of the DECOS integrated architecture developed at Vienna University of Technology. The Linux-extension RTAI allows the execution of tasks in a real-time mode as kernel tasks. LXRT is an addition to the RTAI framework that allows the execution of real-time tasks in the userspace. RTAI and LXRT is deployed in the DECOS integrated architecture.

In this thesis we want to validate whether the Linux-RTAI-LXRT fulfills the requirements to be deployed as an EEE for the DECOS integrated architecture.

1.2 Objectives

The main objective of this thesis is to validate the Encapsulated Execution Environment (EEE) of the DECOS system under test. Based on the results of the executed experiments, assumptions on the correctness can be made.

The diploma thesis objectives can be divided into four parts:

- Development of a tailored SWIFI framework for the validation of the EEE
  An existing SWIFI has to be extended to allow the injection of software faults into a node under test. The SWIFI shall allow the defined, repeatable execution of software fault injections.

- Validation of the DECOS fault hypothesis for SW faults
  To accomplish this, a sound set of test cases for faults stated in the DECOS fault model have to be developed. The test cases cover the aspects of temporal correctness with respect to the integrated application tasks as well as the capability to isolate faulty jobs so they can not affect the correct behavior of the other tasks integrated in the computational subsystem (node).

- Validation of Linux-RTAI-LXRT as DECOS EEE
  The validation whether the Linux-RTAI-LXRT implementation fulfills the requirements to be deployed as an EEE for DECOS will be derived from the performed test cases.

- Design considerations for further improvements of the EEE encapsulation mechanisms of the Linux-RTA-LXRT
  Based on the experiment results, it can be analysed if and where the design of the integrated architecture can be improved or extended to fulfill the requirements of the hypotheses about faults in the system.

1.3 Organization of this thesis

Chapter 2 presents generic aspects of real-time and fault-tolerant systems and requirements of such systems. The deterministic RTAI-LXRT framework on top of a Linux operating system is presented in this chapter. It also gives an overview of validation by means of software fault injection. Several existing software fault injection systems are presented as well.

Chapter 3 describes the DECOS architecture, conceptually a distributed real-time system extended for deployment in safety critical environments. The core communication subsystem is handled by the Time-Triggered Ethernet (TTE).

Chapter 4 presents the basic design concept of the Software Implemented Fault Injection (SWIFI) framework. An existing framework is extended in this thesis to cover the emulation of different software faults in the DECOS environment. Furthermore, specific DECOS implementation details of the DECOS prototype implementation using the Linux-RTAI-LXRT are presented.

Chapter 5 gives the list of test cases for the validation of the fault hypothesis of the DECOS integrated architecture. The results of the executed fault injection experiments are presented as well.

Chapter 6 presents the analysis of the experiment results. Should any experiments not show the correct behavior, proposals for design improvements are given.

Chapter 7 closes with a summary on the results, the usability of the current implementation, and whether it is ready for deployment.

2 Review of the state of the art

In this section, an overview on real-time systems, real-time operating systems, and fault tolerance concepts is presented. Furthermore, the technique of fault injection and existing fault injection tools are introduced.

2.1 Real-time systems

This diploma thesis is about the validation of a component of a real-time (RT) computer system, so we will start with the definition of the term RT computer system.

According to [1], a *real-time computer system* is a computer system, whose correct system behavior not only depends on the *correctness* of a calculation. It is also required that the correct calculation is available at a specified instant of time. This behavior of delivering the calculation at a specified instant is called *timeliness*. The specified instant — the *deadline* — is derived from the *real-time system* where the computer system is integrated.

The real-time system typically consists of a RT computer system, a *controlled object* and an *operator* [1]. Each of these three *subsystems* is called a *cluster*.

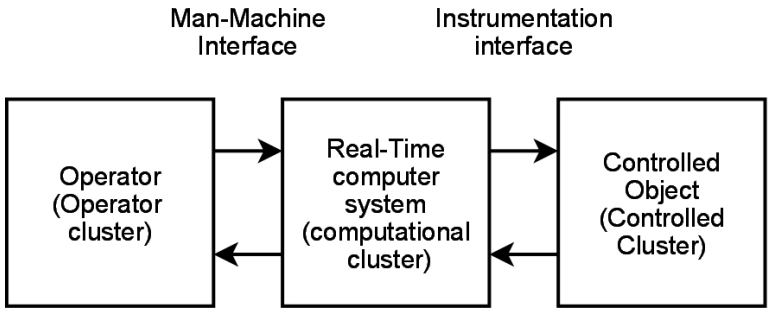

Figure 1: Real-time systems [1]

The following chapters shall present an overview of real-time computer systems aspects and limits with respect to non real-time systems.

2.1.1 Classification of real-time systems

A classification of real-time system can be made upon the specified deadline importance. The deadline of a real-time system is derived from the controlled object.

There are three classes of deadlines we can distinguish. The classification of those categories is made upon the outcome of the miss of the specified deadlines [1].

Soft deadline - A deadline is classified as a soft deadline, if the result of the calculation is of utility even after the miss of the specified deadline.

Firm deadline - For a firm deadline, the result of the calculation has no utility after the deadline has passed.

Hard deadline - A hard deadline denotes a deadline, whose miss can result in a catastrophic event.

Based on the classification of the deadlines, we can classify two types [1] of real-time systems accordingly:

Soft real-time systems

A real-time system is called a soft real-time system, if no single hard deadline exists in the real-time environment [1]. In soft real-time systems, the miss of the deadlines has only minor impact on the result, or the quality of service of the application is only of minor quality [31]. Nowadays RT systems are integrated in a broad range of services and devices. Even consumer multimedia devices are driven by such instead of the classic dedicated digital control approach.
An example of such a soft real-time systems is a real-time video broadcast systems, where the miss of a deadline — the timely transmission of a video frame — causes only slight distortions in the resulting video.
In a digital VCR, the real-time part covers the recording of TV broadcasts at a given time. Evidently, the miss of the deadline (i.e., the start of the TV show) has no catastrophic result, because the only problem resulting would be the miss of the first few seconds or minutes of the program.

Hard real-time systems

A real-time system is called hard, when at least a single hard deadline exists. Hard real-time systems have to meet their deadlines under *all specified load and fault conditions* [1].
For example, operating systems in nuclear power plants have to fulfill much higher demands on the service they provide. The reaction in the case of a problem in the reactor core requires a tight synchronized, timely correct sequence of actions to move the control rods in a safe position in order to stop

the nuclear reaction. A miss of the deadline dictated by the nuclear reaction inside the core can result in a catastrophe costing the lives of many. This leads to the necessity of a hard real-time operating system. Of course, besides other measures such as redundant subsystems, secondary power supplies for the control system, diversity, voting mechanisms, etc.

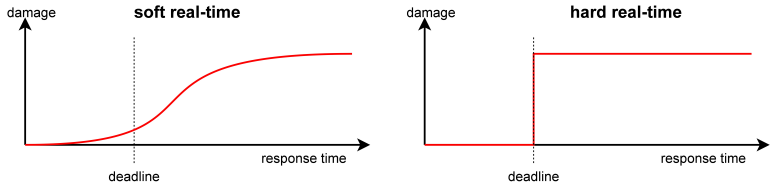

Figure 2: soft vs. hard real-time [31]

2.2 Real-time operating systems

Operating system required for RT systems are called *real-time operating system (RT-OS)*. They must ensure a *predictable service* to the application tasks to be executed on the node. RT-OS have therefore to provide the following services [1]:

- Task management
- Time management
- Interprocess communication
- Error detection

2.2.1 Task management

The Task management service ensures the correct initialization and timely correct execution of tasks in a node. In real-time environments, it is required to schedule multiple application tasks with respect to their defined timing constraints on the available CPU time. Of course, it would be desirable to have an exclusive CPU core for every application task, but economic considerations circumvent such systems. It would simply be a waste of resources as the utilization of the cores would in most cases tend toward zero.

To schedule the existing workload on the available CPU resources, a scheduling plan has to be developed that each task meets its specified deadline, so that the claim of timeliness in the system is achieved. It is a must for the

validation process to adduce evidence for the timing constraints of the whole system.

For the development of the proper scheduling mechanism, we have to distinguish whether a real-time operating system has to handle *Time-Triggered (TT) tasks* or also *Event-Triggered (ET) Tasks*. ET tasks are triggered when a certain event in a systems (i.e., the rise of an observed temperature above a defined limit) occurs and can therefore not be predicted. This makes it difficult to design a system with a guaranteed response time, because events can occur very often and also simultaneous. In systems with a single CPU, events have to be queued and processed in a well defined temporal order: either based on a *first-in-first-out* strategy or based on a priori defined priorities.
Time triggered tasks are easier to handle, when the occurrence of the task, the schedule, is defined at design time. Each task gets a guaranteed exclusive CPU time (i.e., a time slot) to process the workload according to a fixed schedule configuration.

As the main concern of this thesis is about the DECOS environment, where we have to fulfill the scheduling of TT tasks, the following shall give an insight of mechanisms for such time triggered architectures.

Static scheduling

If a system is designed with static scheduling, the only source of events is the *periodic clock interrupt*, which triggers the scheduler. The scheduler has to be configured with a scheduling table which is free of conflicts (i.e., no two (or more) tasks have to be executed at the same time slice). Scheduling information in operating systems is stored in the OS's *dispatcher table* [1]. If the conflict-free schedule is guaranteed, then the scheduler's only function at runtime is to pick the right task from the dispatcher table — a search problem — and activate that task. The actual instant of time can be seen as the input of the search-function. In order to achieve the correct scheduling of dependent tasks in a distributed system, a tight synchronized understanding of the current time among all participant nodes (i.e., the global time) is required. Adequate clock synchronization strategies will be presented later in this chapter. It is also clear that to configure a design-time schedule, which depends on the timing constraints of the scheduled tasks (the deadlines), it must be possible to get the *worst case execution time (WCET)* of all scheduled tasks.

According to [22], the static analysis of real-time programs is only possible, if there are:

- no unbounded control statements at the beginning of a loop,
- no recursive function calls in the task code, and
- no dynamic data structures are used.

As the static analysis of the tasks runtime is required for static scheduling, these three assumptions have to be taken into account when designing application tasks.

To check the schedulability of all tasks in a real-time operating system, we have to know the WCET c_i of all tasks t_i and their respective schedule period P_i. On a single processor system, the schedulability of n tasks can be tested if the following function succeeds [1]:

$$\mu = \sum_{i=1}^{n} \frac{c_i}{p_i} \leq 1; 1 \leq i \leq n$$

2.2.2 Time management

Time

To establish a temporal relationship of events in a real-time computer system, a function to compare instants has to be provided. The metric for time measurement is the physical second. Time is used to establish temporal and causal order in a real-time system. Time can be modeled as *directed timeline* with an infinite set {T} of *instants* [1].
Temporal order is meant to put two instants p and q of instants {T} in relationship, where either p precedes q, q precedes p, or p and q are simultaneous. The ordering of instants on a timeline is called temporal order. Furthermore, if {T} is a dense set, then there is always an instant q between the instants p and r if p and r are different instances [44]. *Causal order* addresses the causal dependencies among two or more events e_j. Causal dependency is given, if an event e_1 occurs before the event e_2, and a variation of e_1 implies a variation of e_2 [45]. *Delivery order* addresses the ordering of the arrival of events in a distributed system. If all nodes see the same ordering of events in a distributed system, then delivery order is given [1].

Clock

Clocks are used to measure the progression of time. A clock consists of

an incrementing counter and an oscillation mechanism generating a periodic event — called *microtick* of the clock — that increases the counter value. The time duration between two consecutive clock ticks is denoted as the *granularity*. Clocks in embedded computer systems are usually triggered by an crystal resonator. Crystal resonators are no *perfect clocks*, as they have have a *drift rate* ρ. The drift rate is defined as the deviation of microticks between a *reference clock* and a real clock per clock ticks of the reference clock [1]. The drift rate of crystal resonators is varying depending on aging and ambient temperature. To compensate the faults of real clocks introduced by the drift rate or clock counter faults, synchronization to external reference clocks is performed. Three characteristics of clocks are of special interest [1]:

- **Offset -** The offset at microtick i is calculated as the time difference between the microticks of two clocks and is measured in microticks of the reference clock. The compared clocks must have the same granularity.
- **Precision -** The precision Π_i is the maximum offset of any two clocks in an ensemble of clocks at a specified instant i. The precision Π is the maximum of Π_i over a *period of interesting microticks* i.
- **Accuracy -** The *accuracy at the instant* i *of a clock* k is the offset of a the clock k compared to the reference clock z *at the instant* i ($accuracy_i^k$). The *accuracy of a clock* k is the maximum offset compared to the reference clock z over a *period of interesting microticks* i ($accuracy^k$)

Global time

In a distributed system with a number of embedded computer systems (nodes) it is required to establish a global time to allow statements on ordering of events. If in a system of n nodes, all nodes are synchronized with a precision Π, then a subset of microticks of a clock k can be defined as a tick of the global time (*macrotick*), and the global time is called *reasonable*. The maximum *synchronization error* for a reasonable global time is bound to ≤ 1. As clocks can not be synchronized perfectly, there is no way to get a better result [1].

Dense time vs. sparse time

In distributed systems, events are often observed at different nodes. The events are timestamped with the global time at the nodes where they are observed. In Figure 3 we can see the problem that the global time can not per se be used to reason on the temporal order and the exact time difference

of events.

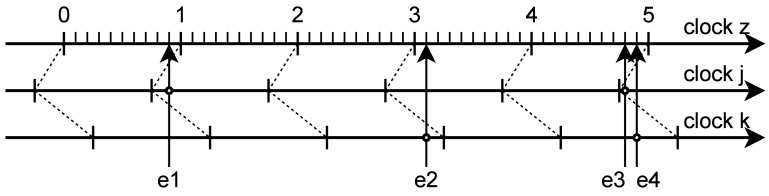

Figure 3: Events with difference of one macrotick [1]

The events $e1$ and $e3$ are observed at node j with the local clock c_j, whereas the events $e2$ and $e4$ are observed at node k with local clock c_k. The clock c_z is the reference clock. Node j timestamps the event $e1$ with the global time 1, and node k timestamps the event $e2$ with 2. Although the real time difference is 22 microticks, the measured value w.r.t. the global timebase is 1 macrotick.

Regarding the events $e3$ and $e4$ we can see that although $e3$ occurs before $e4$, the global timestamp denotes that $e4$ with a global timestamp of 4 occurs before $e3$, whose global timestamp is 5.

According to [1], the temporal order can only be reconstructed if two events differ by at least two ticks of the global time (*macroticks*), as the synchronization and digitalization error is always less than 2 granules of the global time (*macrogranules*).

The requirement on temporal ordering of events can be satisfied by a *sparse timebase*. In a sparse timebase, events are restricted to occur at specific intervals, called *active intervals*, whereas a timebase is called *dense*, if no such restrictions are specified. During the interval between two consecutive active intervals no events are allowed to occur. This implies that the events have to be in the sphere of control of the system, i.e., the computer system [48]. The events generated on the participant nodes at the same global time will occur within the interval π, whereas during the interval Δ no events are to be generated. A timebase with an active interval π of length ϵ and a duration of silence Δ is called ϵ/Δ-*sparse* [1]. To allow the temporal ordering of events, a global timebase must be at least 1/4 sparse [1].

Time management in RT-OS

The Time management in RT-OS shall provide time services to the application tasks, i.e., clock synchronization and activation of tasks at a specified

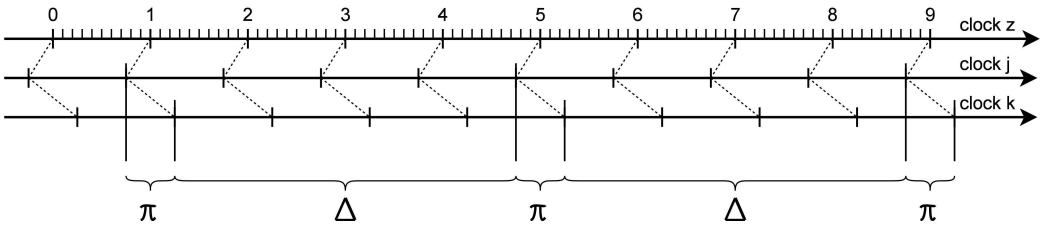

Figure 4: Sparse timebase [1]

instant of time [1].

Clock synchronization

Clock synchronization is the process of making different units in a computer network to agree on a same timeline.
Clock synchronization algorithms aim at bringing a set of Distributed clocks into agreement. *Internal clock synchronization* is the action of bringing all clocks within a local network into a closer agreement to each other while *external clock synchronization* aims at bringing a clock or a set of clocks into agreement with an external reference time [1].

Internal clock synchronization

Internal clock synchronization tries to establish and maintain synchrony among a set of clocks such that the maximum deviation between any two clocks can be bounded by a constant value denoted as precision Π [1]. The clock synchronization precision represents the maximum offset between any two clocks in the system.
The clock synchronization depends on the jitter introduced by the network (message transmission jitter), the drift rates of the oscillators, the clock synchronization interval, the jitter of processing the clock synchronization, and the clock synchronization strategy and mechanisms.
Internal clock synchronization is a periodic activity that is executed at the nodes of a distributed system that perform the following three steps:

1. Read the values of the other clocks.
2. Calculate the clock state correction term based on a set of remote clock readings.
3. Apply the clock correction term to the clock.

Based on the clock correction policy, clock synchronization algorithms can be classified into

- clock state correction,
- clock rate correction, and
- combined approaches.

External clock synchronization

External clock synchronization is the process of establishing a synchronization between a set of clocks and an external time source (e.g., a GPS receiver) [1]. It is the goal of external clock synchronization to synchronize a set of local clocks to external reference time such that the maximum deviation between any local clock and any reference time server can be bounded by a known and constant value.

Time services

The Time Services in RT-OS are crucial for the temporal order of events in a node. According to [1], the basic time services cover the provision of temporal ordered sequences to activate the TT tasks, the ability to send the messages on the communication medium at a specified instant in the future, and the *time stamping* of events that arise at a arbitrary instant. Furthermore, it shall provide a function to convert the local time to the *wall clock time* (Gregorian calendar).

2.2.3 Interprocess communication

In RT-OS, tasks are not always independent from each other. For two or more tasks to access a common region of data it is sometimes required to use synchronization mechanisms to ensure consistent views on that data. In *commercial-off-the-shelf* (COTS) operating systems this is mostly done by using semaphores. Semaphores cause a processing overhead which introduces additional delay by context switching of tasks due to blocking [1].

For TT tasks with a static schedule, the tasks can be ordered in a way that tasks do not interfere each other, whereas for ET tasks a solution would be to duplicate the common data and access the copy during execution. After execution, the data can be written back to the common region [1]. This way a blocking behavior can be avoided.

2.2.4 Error detection

RT-OS have to implement mechanisms to monitor the correct execution of tasks and detect faults. Faults can encounter in the temporal or value do-

main [1].

Faults in the temporal domain can be detected by monitoring the execution times of tasks and comparing the timing characteristics to the statically analyzed WCET. When a task exceeds the WCET, the RT-OS has to terminate the corresponding task and mark the task as faulty.

In the value domain, faults can be detected by mechanisms like the double execution of tasks. For the double execution of tasks, the time required for the two consecutive executions is the double of the WCET. This time slot has to be provided by the RT-OS. After the execution of the two calculations, the RT-OS can compare the results and decide whether the result is correct or not. The decision whether a task shall be executed twice, or not, is to be done at design-time by the developer of the application task [1].

2.2.5 Requirements on real-time operating system calls

A real-time operating system has to provide a time predictable service to all the application tasks to allow an exact statement about the timing constraints to achieve the claim of timeliness. Therefore a RT-OS has to make a compromise about the *application programming interface (API)* it provides to the application designer, as not all API functions available (e.g., desktop operating systems) have a definite, bound timing behavior [1].

Dynamic memory allocation

At a certain point of time after successive allocations and deallocations, when memory gets more and more fragmented, the function has to perform an indeterministic, unbound number of extra CPU cycles to find a memory block of the requested size. In the worst case, the memory chunk of the requested size can not be found, and the function will return an error.

Dynamic memory deallocation

The actual release of a memory block causes several consecutive functions in the OS to be called. In security related OS, the freed memory block gets zeroed [21] to prevent security holes before merged to the adjacent free memory regions, which have to be found by a search function. A design consideration on bound execution times on dynamic memory allocation at runtime can be seen in [21].

Recursions

The use of recursions, which depend on variable input can have dramatic

effects in term of stack usage. Each recursion allocates additional stack space, which after a finite number of consecutive cycles will result in a stack overflow. If recursions in real-time tasks are required, the designer of the task has to achieve counter-measures like limiting the recursion depth, checking the stack space left and ensure an upper bound for the execution time [22].

Infinite loop detection

Tasks executing infinite loops due to incorrect inputs or program faults cause deadline violations. A mechanisms for detecting deadline violations has to be provided by the RT-OS. Tasks suffering from such a fault shall not affect the operation of other tasks.

2.2.6 RTAI - Real-Time Application Interface for Linux

RTAI is an extension to Linux for real-time scheduling of application tasks. It was originally designed at the Politecnico di Milano for projects in the aerospace domain [19].

Linux is per default not capable to fulfill the timely claims on real-time systems. The scheduling mechanisms of plain Linux are not sufficient to fulfill the required timeliness of hard real-time systems. To overcome this limitation, RTAI patches the kernel and injects an additional subsystem, the ADEOS nanokernel, into the hardware abstraction layer (HAL) to intercept the default interrupt handling and replace this mechanism with RTAI kernel code. The reason behind is to allow the ADEOS nanokernel full control over the timers of the computer systems: The 8254 chip with the four timers available to operating systems [20]. This small modification in the kernel code allows to run the Linux tasks on top of the RTAI framework in the userspace. In fact, it allows to run several OSs on top of RTAI, each within a special, encapsulated domain. Each time an interrupt arises, the ADEOS kernel as the now default interrupt handler of the timer intercepts the call and routes it to the affected domain along the interrupt pipeline.

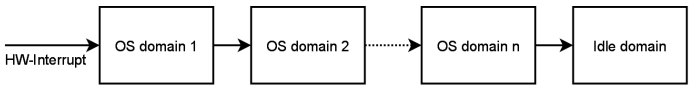

Figure 5: ADEOS interrupt pipeline [31]

RTAI modules

RTAI is a modular architecture, where all functions are encapsulated in

specific modules. A system designer can choose among the existing function blocks and tailor the system to his specific needs. By selecting just the required modules, it is possible to design a real-time system with a very small footprint of only several megabyte. RTAI [20] consists of the following modules:

`rtai:` The `rtai` module is responsible for the instantiation and initialization of all RTAI variables and objects required for proper functionality. As RTAI replaces the default LINUX interrupt handlers, it has to make a copy of the *interrupt descriptor table*, the assignments from the 256 different interrupts to the corresponding Linux-handler, and initializes the computer's underlying interrupt management.

`rtai_sched:` Scheduling in RTAI is achieved by the RTAI scheduler module `rtai_sched`.The external trigger event to activate this module is created by the computer's 8254 (or equivalent) PC Timer IC. Tasks get scheduled based on whether they are in the ready-queue and on their priority in the system, whereas the Linux-OS has the lowest priority.

`rtai_fifos:` This module is responsible for the communication with the RTAI primitive *FIFO*. FIFO allows to build a communication path among RTAI tasks, but also from an RTAI task to a Linux task. An RTAI-FIFO is seen as a standard character device from the non real-time Linux tasks and is implemented as non blocking.

`rtai_shm:` Shared memory operations in RTAI are achieved by the `rtai_shm` module. The module presents functions for the creation, write, read and destruction of the shared memory object. Shared memory regions can simultaneously be used from RTAI tasks, from the kernel and from Linux tasks in userspace.

Real-time tasks in kernel-space Standard RTAI tasks are modules running in the kernelspace. This allows quick task switching times, as no mode switch from kernelspace to userspace is required. RTAI tasks have the same structure as standard modules.

Tasks real-time behavior in RTAI can be ensured in two ways:

`one-shot mode:` The first possibility is to schedule a task at a fixed instant in the future. To accomplish this, RTAI uses the timer function in *one-shot mode*, so the timer interrupt fires once. This allows to define an instant in the future, but no periodic tasks. After initialization of the

timer, the task has to be put in suspended mode. The timer interrupt wakes the job, so it executes at the defined instant [19].
To re-launch a task in the future, the timer function has to be re-enabled, causing additional overhead.

`periodic mode:` The second option is the periodic event of the timer. This mode is set up with a period between two occurrences of the timing interrupt. Each occurrence of the timer de-blocks the former suspended task and executes it. No additional initialization during runtime is necessary.

Real-time tasks in userspace (LXRT)

The approach to run modules in kernelspace has several disadvantages, like the problem of limited debugging mechanisms, or that a faulty RTAI task can lead to a system crash. Another drawback of kernelspace modules is that it is not possible to use dynamically linked libraries.

As a solution to these problems, the RTAI extension *LXRT* was added to the framework. This extension allows RTAI tasks to be run in the userspace, where the existing memory protection mechanisms reduce the possibility of system crashes.
The mechanism is implemented by using a *buddy*-task in the kernelspace for each LXRT-task in userspace. The buddy-task acts as a server for the LXRT-task: each RTAI function called from the LXRT-task gets routed to the buddy-task and performed there. LXRT tasks can be run as either hard or soft real-time tasks. This behavior can be changed at runtime [20].

LXRT is mostly used to test new code. After successful tests in the userspace, and if no dynamic linked libraries are required, they can easily be transformed to kernelspace modules, where the additional overhead, introduced by the communication with the buddy task, can be removed.

2.3 Fault tolerant systems

This section presents the aspects of fault tolerant systems. In safety critical applications, systems must be able to tolerate the presence of faults. For example, in the aerospace domain, a single fault in the computer system may not affect the operation of the whole system. Even in the presence of faults, the pilot depends on the services of the system to land the aircraft in

critical situations. We will start with the definition of the terms *fault*, *error* and *failure*.

2.3.1 Faults, errors and failures

In fault tolerant systems, it is of utmost importance to prevent a single fault in a subsystem to affect the correct operation of the whole system. It has to be a design goal to detect the abnormal operation of such subsystem at the smallest level possible. To specify the mechanisms, this section will at first give the definition of the terms *fault*, *error* and *failure* to understand the dependencies among them.

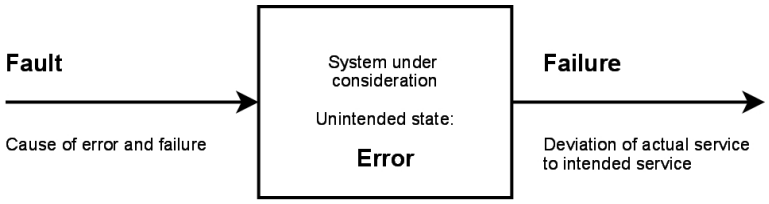

Figure 6: Faults, Errors, Failures, taken from [24]

Faults

A fault is the cause of an error. An erroneous system state can lead to a system failure. Failures are events, whereas faults and errors are states [1]. In computer systems, faults can originate in both hardware and software. A fault can be the missing check for division by zero (software fault) or incorrect sensor readings due to faulty connection (hardware). According to [24], faults can be classified by five categories:

Fault nature - If the fault originates by intention, (i.e., someone short-circuited a connection or injected a malicious code), we speak of an intentional fault. If the fault occurs by coincidence like a loose connection, the fault is called a chance fault.

Fault perception - If the fault originates by design (i.e. the division by zero in a code fragment), the fault is called a design fault. Should the fault originate in the physical domain, for example a stuck memory cell in the computer's RAM, it is called a physical fault.

Fault boundaries - This categorization is based on whether the fault is caused by an event inside the system, for example the previously stated stuck memory cell, or the event is triggered by some external action like

mechanical stress due to vibrations causing loose connections. Faults can therefore be defined as internal or external faults.

Fault origin - Fault introduced during the development phase, like incorrect coding of algorithms are called development faults. In contrast to development faults, faults originating in incorrect interaction during the operation are called operation faults.

Fault persistence - Faults appearing occasional and disappearing without repair or maintenance are called transient faults. A node periodically reading and processing sensor values over a loose connection reads correct and incorrect values depending on whether the connection is good or bad. Evidently, when the connection is good, the correct value processed overwrites the faulty value, meaning the error disappears until the connection breaks up again. If the connection is permanently broken, the controller cannot read correct data and the node will permanently calculate invalid results. The fault can only be removed, if maintenance is done on the connection. This type of fault is called a permanent fault.

Classification of software faults

Software faults can be classified into the following typical categories due to their fault behavior:

Bohrbugs - Bohrbugs are software faults that can be reproduced when repeating the operation on the testing components. This makes it simple to deal with them by software fault detection mechanisms in the test phase [25]. Due to their solidness, the name Bohrbug was derived from the Bohr atom. Removing Bohrbugs in systems already deployed to the field is simple, as the fault can be reproduced in the environment at the developer's site when the input parameters are known [26].

Heisenbugs - Heisenbugs are much more complicated to find, as they tend to not always produce a fault when repeating the operation with the same input data. Bruce Lindsay says about Heisenbugs: *"Heisenbugs as originally defined [...] are bugs in which clearly the behavior of the system is incorrect, and when you try to look to see why its incorrect, the problem goes away [27]."* Heisenbugs appear in rare cases and can be pinned down to either strange hardware conditions (e.g., device fault), limit conditions (e.g. lost interrupt) or race conditions. [28]

Software aging - Software aging is a problem of certain software systems, when running for a long time: the rate of fault-occurrence rises during runtime. The solution to software aging in most cases is to restart the component, called *rejuvenating*. The failed interception of a Scud rocket in Dhahran during the Irak war in 1991 was due software aging: The precision of the targeting system of the Patriot rocket decreased with increasing runtime, and the suggested rebooting process to overcome this problem was not performed [26].

Errors

As we can see in the Figure 6, an error is the result of a fault. In computer systems, an error is an internal state. Errors can be classified into two categories:

- Transient errors
- Permanent errors

If a fault on the input of a task arises, and the application task's output is directly calculated from the input, then the error is classified as a transient error. As soon as the fault is repaired, the application task will calculate a correct result and the error state disappears [1].

Should the application task's result not only depend on the actual input, but the calculation also depends on a past value of an input, then the error is persistent [1]. A typical example is a temperature controller whose calculation is also based on past temperature sensor readings. The past sensor readings are stored in the tasks *history state (h-state)* and are used for future calculations. The incorrect h-state persists until a "repair" action is performed (i.e., a reset of the h-state) [1].

Failures

"A Failure is an event that denotes a deviation between the actual service and the specified or intended service, occurring at a particular point in real time [1]." Failures can be classified into the following 4 classes [1]:

Failure nature - Failures can occur in either the temporal or the value domain. Failures occurring in the time domain (i.e., results are presented outside the specified timing window) are called *timing failures*. *Value failures* are failures which manifest in an incorrect value at the system interface.

Failure Perception - If a failure presents itself to all connected interfaces consistently, the failure is called *consistent failure*. If the failure is not consistent at all interfaces, the failure is called *inconsistent failures*. A *Byzantine failure* is a typical inconsistent failure.

Failure Effect - The effect a failure has on the system can either be *benign* where the costs resulting are minor or *malign*, where the costs of the failure are much higher than the system costs. Systems able to tolerate malign failures are called *safety-critical systems* [1].

Failure Oftenness - A failure occurring only once in a specified time interval is called a *single failure*. A *permanent failure* is a single failure disappearing after a repair action. Failures occurring a repeated number of times in a specified interval are called *intermittent failures*.

2.3.2 Fault tolerant system characteristics

The requirement for the system behavior in the event of a failure can be derived from the environment it is implemented into. When detecting abnormal operation, it is required to put the system in a condition, where no harm is possible to the system and its environment. If such state — the *safe state* — can be identified, the system can be implemented as a so called *fail-safe* system [1]. An example would be an elevator control logic: In case of a fire in a building, a safe state for elevators is to direct all cabins to the ground floor and open the doors. For a failure inside the elevator logic itself, this state would be to simply stop all cabins at its current position, regardless if they are between two levels, doors open or closed.

There are applications, where such safe state can not be reached (e.g., control system of aircrafts). For such scenarios, advanced error handling is required to provide a *minimum level of service* [1] to continue the flight and allow the landing of the aircraft whether automatically, or manually by the pilots. These systems are called *fail-operational* and must allow the presence of faults during operation.

Fail-safe system - If in a system a safe state can be identified and reached quickly upon the occurrence of a failure, the system is called *fail-safe* [1].

Fail-operational system - If in a system a safe state can not be identified but it provides a minimal level of service to avoid a catastrophe even in the case of a failure, the system is called *fail-operational* [1].

The ability to tolerate faults during the operation classifies systems as *fault tolerant systems* [1]. In fault tolerant systems there are two levels where fault tolerance can be implemented:

- systematic fault tolerance
- application-specific fault tolerance

A systematic fault tolerance is implemented at the architectural level. For the system to detect faults, the system must produce the same calculation result for the same set of input data. This behavior is called *replica determinism* [1]. From this behavior the system can reason on faults in the temporal and spatial domain by replication and detect the fault by comparing the results.

Application-specific fault tolerance is achieved by the application level of the system [1]. The functions for detecting discrepancies of results is implemented in the application code.

2.3.3 Fault tolerant units

The purpose of *fault tolerant units (FTU)* is to encapsulate a fault in a well defined region to prevent a propagation of the fault in the system. There are several methods to realize fault tolerant units, depending on the behavior of a unit (node) on the occurrence of failures.

Fail-Silent nodes

If a node is implemented such that the result of a calculation in the presence of a fault is either a *correct* result or *no result* at all, then a node is called a *fail-silent node*. The mechanism to achieve fault tolerance is therefore to duplicate the functionality of the node in another node. During normal operation the nodes provide identical results. In the event of a failure, only one node produces a result, the correct result [1]. To reestablish redundancy in the event of a failing node, a *shadow node* can be implemented which has the same functionality and activates itself only after the detection of a failing node.

N-modular redundancy

Nodes which are not implemented as fail-silent nodes can exhibit a result even in fault scenarios. It is therefore not possible to derive the correct calculation from just two nodes, as both will generate a result. A *voting mechanism* has to implemented which accepts the result from the majority

of results as the correct one to gain a correct result. In order to tolerate m failing node, $n = 2 * m + 1$ nodes have to be implemented. The voting mechanism can either be an *exact voting*, where the result must be exactly the same on the majority of nodes, or an *inexact voting*, where the results can be within an acceptable range, the *application-specific interval* [1].

Byzantine resilient fault tolerant unit

For nodes whose fault mode cannot be determined, a Byzantine resilient protocol has to be established. To tolerate k byzantine faults, $(3k+1)$ nodes are required on executing a Byzantine resilient protocol [32]. Each node is connected to each other node by separate communication paths, and $(k+1)$ communication rounds must be performed.

2.3.4 Fault hypothesis

Tolerated faults in a fault-tolerant system are specified in the fault hypothesis. A fault hypothesis specifies the type, number and frequency of faults that should be tolerated. A Fault hypothesis of a fault-tolerant system contains the following items:

- The smallest unit of failure in fault-tolerant systems is defined a *Fault Containment Region (FCR)*. A fault containment region is defined as a set of components that is considered to fail as an atomic unit, and in a statistically independent way with respect to other fault containment regions [33].

  Fault tolerance is usually achieved by using replicated FCR. In case of one FCR has failed, the operation of the redundant FCR should maintain the functionality, while the failed FCR can perform recovery actions after the fault has disappeared.
  Faults in one FCR of a distributed fault-tolerant system shall not be able to propagate and to interfere with the operation of non-faulty FCRs.

- Fault model - describes the type, the number, and the fault arrival rate of FCR failures that should be tolerated.

  - The type of faults in the fault hypothesis describes the type of faults an FCR may suffer (permanent or transient). Failure types can be classified into three modes of FCR failures based on the effect they show in the interfaces of other FCRs: detectable, undetectable consistent, and inconsistent.

An FCR delivers a detectable erroneous service, if it either delivers no service, or delivers an incorrect service that can be consistently detected by other FCRs.
An FCR suffers from an undetectable consistent failure, if the service it delivers is incorrect, but the failure is not detected by other FCRs and this service is consistently received by other FCRs. For instance, fail silent violations in the value domain in a TTP/C system are undetectable consistent failures.
An FCR is said to fail inconsistently, if it shows different fault semantics to different FCRs (e.g., a Byzantine failure).

  - The number of faults in the fault hypothesis describes the number of FCRs that may be faulty at a time without affecting the functionality of the system. The number of tolerated faulty FCRs depends on the level of the deployed redundancy. For example, to tolerate x consistent faults of the communication channels, there must be at least $x + 1$ communication channels [43].
  - Fault arrival rate defines the temporal distance between the point in time when one FCR becomes faulty and the point in time when another or the same FCR becomes faulty (having in mind that only transient faults are considered), without compromising the functionality of the system. The fault arrival rate depends on the duration of a fault that affects an FCR and the time that the FCR needs to recover from that fault.

- The *Never-give-up strategy (NGU)* is a mechanisms that defines the behavior of the system in case that the system faults that are outside the fault hypothesis. Such faults cannot be handled by the fault tolerance mechanisms of a system and therefore, the system will fail. If faults have a transient nature (e.g. transient EMI disturbances), the system can recover by initiating a restart. The TTP/C protocol for example, detects the violation of fault hypothesis by means of the membership and the clique avoidance algorithm [46].

Any fault-tolerant system must be designed and evaluated against an explicit fault hypothesis [42].

2.4 Fault injection

For safety-critical applications, fault tolerant computer systems must be implemented. Such systems must be able to tolerate specific faults. The presence of these faults specified in the fault hypotheses must not affect the

operation of the system. A fault hypotheses for fault tolerant computer systems specifies several fault classes, which can be divided in hardware faults and software faults.

The validation of a fault tolerant computer system requires the evaluation of the system's operation under all faults specified in the fault hypotheses. As faults are normally rare events, the validation procedure requires an artificial acceleration of the faults specified [39]. The activity to accelerate the occurrence of faults is called *fault injection* [38].

2.4.1 Techniques

Faults injection can be accomplished at several architecture levels [38]. We can distinguish among *simulation based fault injection*, *hardware implemented fault injection (HIFI)* and *software implemented fault injection (SWIFI)*.

Simulation based fault injection
Simulation based fault injection can be done at a very early stage of the development of a system. It is performed on the simulation model of a chip or system design. Depending on the simulation tool, it is either possible to modify the circuitry of the system or force input data to faulty states (electrical or logical). The advantage of simulation based fault injection is that no damage of the system can occur [39]. This technique is widely integrated in design tools of VHDL and FPGA devices.

Hardware implemented fault injection
Hardware implemented fault injection is done on a real systems. The test cases of the fault hypothesis cases are applied to an operational system to observe the results. The disadvantage in contrast to simulation based fault injection is that the system under test can be damaged during the test runs. Testing methods for HIFI are the stressing of systems with electronic, radiation or thermal effects. According to [38], testing methods can be classified in:

- Hardware fault injection with contact where direct physical contact to the system under test occurs. Test scenarios of this class are e.g. the applying of different logical and electrical levels at a pin of an electronic chip. It is evident that experiments where the electric level applied exceeds the specification of the device, irreversible damage to the device can be caused.

- Hardware fault injection without contact if no direct physical contact to the system under test occurs. This includes the stressing of a system with electromagnetic interferences — *electromagnetic fault injection (EMFI)* — or applying radiation to the system. A framework for automated testing of a TTP/C cluster with EMI has been developed by the author during a practical at the Realtime Systems Group at the Technical University of Vienna.

Software implemented fault injection

Software implemented fault injection (SWIFI) is performed by implementing a fault injection (FI) code into the target system. This FI code is used to trigger the required faults at locations not accessible by other fault injection strategies. SWIFI allows the activation of the specified faults at defined instants in a reproducible manner. The advantage of SWIFI is furthermore that the implementation of the FI code is relatively simple, no additional hardware is required, and usually no damage can occur on the system under test. A challenge on SWIFI is to reduce the *probe effect*, which denotes the deviation of the system under test (in presence of the FI code) to a system without the modifications [1]. The injection of the faults in a computer system can be accomplished by different methods [38]:

- Compile-time injection: The fault to be injected is applied by modification of the system's program code prior to loading the program image on to the system. This strategy covers hardware, software and transient faults. The fault is activated at the instant when the system executes the faulty code.
- Run-time injection: The trigger of the fault can be controlled by special events. Mechanisms for triggering runtime-injected faults are:
  - Timeout: The instant a fault gets activated is based upon the interrupt of a timer. A computer timer gets preloaded with a specified timeout, and when the timeout occurs the fault injection code gets triggered.
  - Exceptions/Traps: This methods provides more flexibility, as the code injecting the fault can be triggered upon either a specified execution of code or on events like external activation from a software component (i.e., a SWIFI framework).
  - Code insertion: The fault injection is performed as soon as the execution reaches a defined instruction. Prior to the real instruction, the FI code performs additional instructions to change the system state.

2.5 SWIFI frameworks

This section presents three exemplary existing testbeds for software implemented fault injection (SWIFI). These are the *Fault Injection based Automated Testing environment (FIAT)*, DOCTOR, a SWIFI for distributed real-time architectures on the HARTS platform, and Xception, a framework for the injection of faults in modern processors.

2.5.1 FIAT

The *Fault Injection based Automated Testing environment (FIAT)* - was one of the first methodic approaches to the field of automatic testbeds for fault injection. Introduced in 1988 its goal was to provide an monolithic tool to evaluate distribute real-time systems by testing the *error detection/recovery mechanism (EDRM)* not only indirectly via fault injection but also directly by activating the EDRM and providing statistical data about the experiments [9].

FIAT provides a graphical user interface that supports the test personnel at all stages like the profiling of source code, the definition of test cases, automated test procedures and statistical data mining.
The experiment process relies on several base information that has to be defined before starting the evaluation of the RT-system:

- **Workload**: The workload definition specifies the default application carried out on the system and being tested. The workload gets profiled by the *attribute extractor* that parses the source code of the tasks and gathers some basic information like task-identifier and addresses of code- and data-segment of all tasks. The automatically gained information from these profile runs are stored in special tables called *domains*.

- **Fault classes**: Fault classes are abstract data types specifying all possible fault-injections for the FIAT system. Information stored per fault class is a list of addressable nodes, affected tasks, memory type, size of memory to be manipulated, memory addresses, data pattern and memory manipulation method plus their functions per property how to select values from the extracted domains. An example of [9] is given below:

```
Mechanism:  FaultClass1
Fire:       Firelist        Fire_Selection_Method
```

```
Task:       TaskTable       Task_Selection_Method
Element:    ElementTable    Element_Selection_Method
FaultSize:  ElementTable    Size_Selection_Method
Position:   PositionTable   Position_Selection_Method
Mask:       MaskTable       Mask_Selection_Method
Behavior:   BehaviorList    Behavior_Selection_Method
```

The fault class definitions are processed by the *Fault instance generator* which applies the methods of the fault classes on the domains created by the attribute-extractor and generates a list of *fault instances*. These fault instances can be selected for the experiments being executed by the fault injection mechanism.

- **experiment descriptions**: Experiment descriptions follow an predefined experiment description format and contain the experiment, the database name where to store results, and a syntax how the experiment has to be executed, i.e., a loop with the amount of iterations which embodies the workload definition, a fault injection set consisting of the fault class with the amount of injections and the data collection command.

  This description is transformed by the *experiment description translator (EDT)* to *experiment scripts* specifying the concrete experiments being processed by the *Fault injection manager (FIM)*. Experiments are collections of test runs containing instances of the predefined test classes. An experiment can contain fault class instances of either the same or from different test classes. Multi-experiments combine several experiments to one run.

The system can be divided into two parts, the node triggering the injections with the FIM and the connected nodes where the faults get injected being under the control of the *fault injection receptacles (FIRE)*. The distinct software parts depicted in Figure 7 and Figure 8 are connected by a local area network to exchange commands.

Workload A workload is an *observable* set of real-time communicating tasks [9].

Task A task is an observable scheduled unit of computation communicating through observable communication media named channels [9].

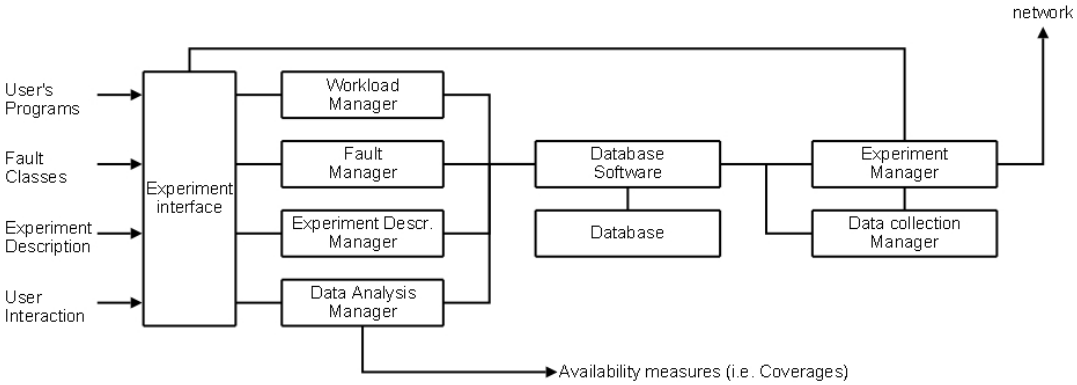

Figure 7: FIAT FIM overview diagram [9]

The interactive FIM has complete control over the test-phase as it starts the experiments and collects test results sent by the nodes' FIRE software layer. Targeted nodes receive their commands from the network at the FIRE. This software layer adds all the functionality at the nodes to start the experiments and deliver test results back to the FIM node.

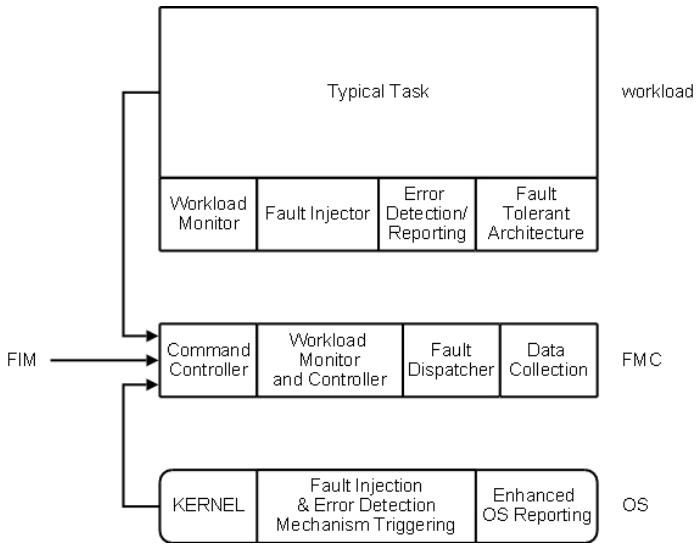

Figure 8: FIAT FIRE overview diagram [9]

The Command controller manages the incoming commands and forwards FI-commands to the node's fault dispatcher. The fault dispatcher has a counterpart in each task involved where the actual injection is done. As the operating system has several protection mechanisms that prevent corrupting the tasks memory from outside the actual injection has to be done by the

task itself. This is the most intrusive component in the system and had to be designed very carefully as it adds a reasonable overhead in terms of processing time (probe effect). The node's data collector acts as a concentrator for all tasks results and delivers the data back to the FIM where being stored in the experiment database for further statistical analysis. All tasks running in the node are under control of the workload manager that decides from the experiment description which tasks to run, suspend, stop, resume or reset after experiment end.

The faults injected shall represent these that have a high probability to appear in real-world use but are very rare to observe. The goal is to significantly reduce the time it would normally take to observe a fault of a dedicated class. FIAT supports this by giving the possibility to inject errors in user tasks at code-level and data-level with manipulations like bit-clear, bit-set, bit-flip, byte/multibyte set/clear and pattern injection within several tasks. The profiling process at compile-time relieves the testing personnel from gathering detailed information about the memory locations as all this information is extracted automatically so the experiments can be defined at a higher level. By giving just abstract definitions of the desired experiments, one can port the FIAT SW-part also to other hardware/software platforms and repeat tests of the same set of fault class instances on another system generation.

As the test results are stored in an relational database with structured query language interface one can compare the uniform stored results among other platform without prior adjustments in data format.

One major drawback in this design is the missing functionality of injecting communication faults or corrupting message data.

2.5.2 DOCTOR

DOCTOR [11] is a fault injection framework for the HARTS [12] distributed real-time architecture. It provides an interactive user interface to plan and define the testing procedure at a higher level. The possibilities of DOCTOR not only cover the evaluation of simple fault scenarios inside a node like memory manipulation but also CPU faults and communication channel faults which is makes it suitable for distributed computer systems [11].

DOCTOR specifies 4 attributes for fault injection tests, namely the:

- **F**aults: the fault classes
- **A**ctivations: the workload on the system under test
- **R**eadouts: the results gained from the system under test, i.e. fault-injection results
- **M**easures: quantitative statements derived from the readouts which give information about the dependability of the system

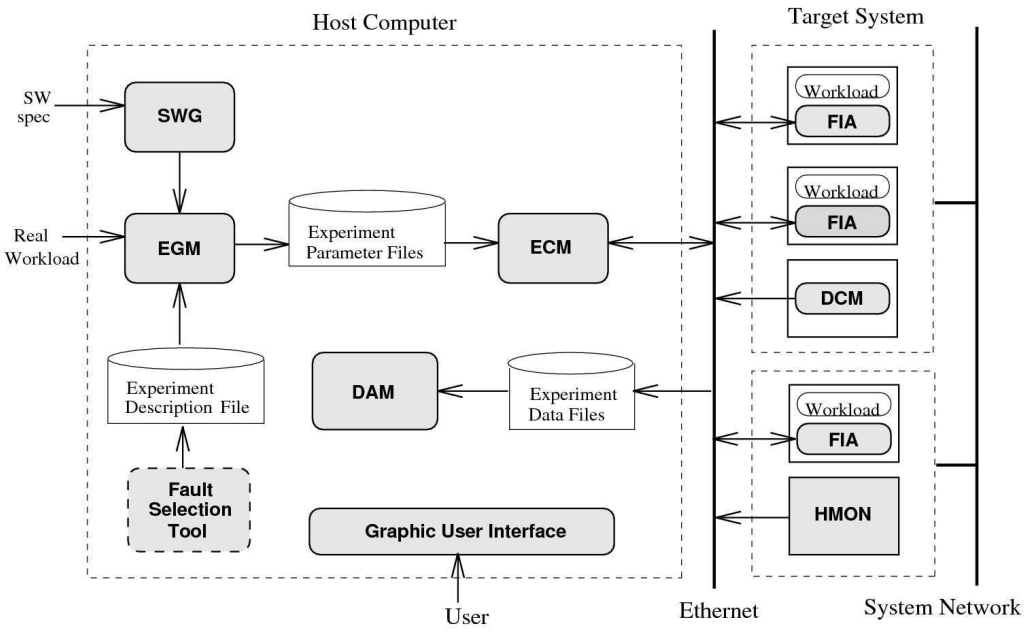

Figure 9: DOCTOR overview diagram [11]

The *experiment generation module (EGM)* generates workloads for all nodes under test and creates a set of experiments — the *experiment parameter files*

— based on an *experiment description file*. The workloads are transferred to the nodes for execution. The *experiment control module (ECM)* controls the actual experiment, i.e., the fault injection, and sends the commands to the nodes under test accordingly [11]. In a node under test, the *fault injection agent (FIA)* performs the injection, and detected errors in this node are recorded by the *data collection module (DCM)* for later analysis.

The host computer, controlling the testing procedure, is connected to the node under test by an ethernet connection. This reduces the adoption to other system implementations to just rewrite the target nodes' fault-injection layer. The software injection mechanism in DOCTOR uses either software traps for injection of transient faults, or compile-time injection for permanent faults [11].

2.5.3 Xception

The SWIFI tool *Xception* [40] is a framework for injection of software faults into modern processors like Motorola PowerPCs, Intel Pentium based processors, or the Alpha AXP architecture. It makes use of debugging hardware, implemented in modern processor generations for the fault activations. These debugging units in advanced processors support the activation of faults either by trigger from an internal timer or upon specified load or fetch instructions. Xception supports the injection of transient faults in a unit of interest inside a processor (i.e., integer unit, floating point unit, data bus, address bus, registers) which can not be reached by many SWIFI frameworks [40].

Results of SWIFI experiments can be gathered by using the internal performance monitoring mechanisms available on the supported processors. These allow to collect data like number of memory read and write cycles, and performed execution cycles after an fault injection [40]. The Xception architecture is depicted in Figure 10.
The *experiment manager module (EMM)* on the host computer presents an user interface to the operator. It allows to define experiments w.r.t. fault location, fault triggers, and fault types (i.e., stuck-at-zero, stack-at-one, bit-flip). Furthermore, the EMM performs the control and result collection. Experiment commands are sent from the EMM to the *fault setup module* on the target system, which routes the command to *the kernel module*. The kernel module performs the actual fault injection. If the system detects an error due to an injected software fault, the generated error is passed from the kernel module to the EMM for analysis.

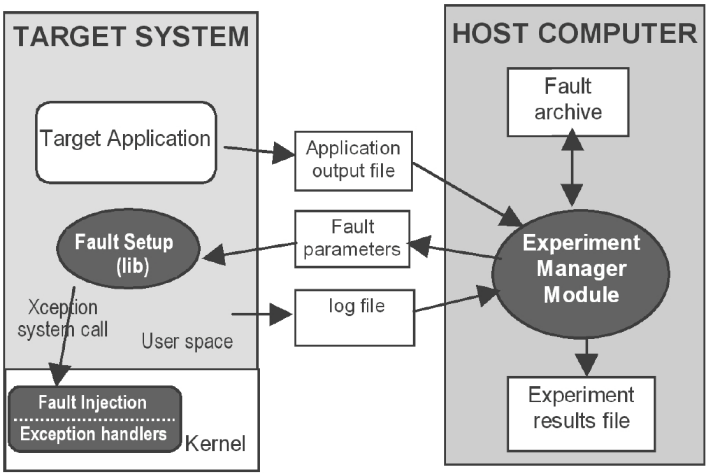

Figure 10: Xception overview, taken from [40]

The architecture of Xception is such that porting of the framework to other processor systems just requires the rewrite of the kernel module code, which encapsulates the hardware information on the debugging and monitoring mechanisms of the targeted processor [40].

3 The DECOS Framework

The Dependable Embedded COmponents and Systems DECOS project is a project under the umbrella of the Sixth EU Framework Programme for Research and Technological Development (FP6). It is an integrated architecture for the automotive, aerospace, and control domain [7].

The DECOS integrated architecture makes use of a Time Triggered Protocol for communication among the connected nodes. At present, there are two implementation variants realized: one with Time Triggered Protocol Class C (TTP/C) and the second one, implemented in the system under test of this thesis, is based on the *Time-Triggered Ethernet (TTE)*.

3.1 DECOS system design

Nowadays, implementations in the automotive domain mostly follow the design principle of federated architectures. In modern cars, a huge amount of sensors, actors, and control units from various suppliers are installed. Communication between those units is in most cases achieved by the *Controller Area Network (CAN) bus*, a serial field bus invented by BOSCH in 1983 with a maximum data rate of 1 MBit/s.
Having a look at current luxury cars we can count up to 90 nodes per vehicle [47]. The disadvantage of such federated systems is the huge amount of wires to connect all those nodes and the multitude of connectors, each of them a source of faults, thus driving up the costs for wiring and testing higher and higher for each additional integrated node.

DECOS takes a different approach to such scenarios. By design the DECOS projects aims at an integrated platform for all intelligent properties inside the domain. The DECOS hardware shall provide a well specified and prevalidated platform to integrate all control units. In contrast to the federated approach, where each application takes a separate computer or microprocessor hardware unit, DECOS nodes allow the integration of various dependable or independent application tasks inside a single node without affecting each other [49, 34].

The whole system with connected nodes, communication medium, and switching unit (TTE switch in the current implementation with TTE) is called the *DECOS cluster* [49]. The framework provides strong encapsulation of subsystems containing dependable applications by the use of "virtual" network routes within the underlying redundant communication channels to circum-

vent the considerable testing effort due the magnitude of integrated applications in a DECOS cluster.

3.1.1 DECOS architecture

In Figure 11 we can see the layered architecture of DECOS. The lowest layer, the base architecture, hides all physical and software design issues not necessary to the knowledge of the developer [49]. These are e.g. the safety critical mechanisms, the used communication medium and the actual implementation of the communication protocol used. This makes it possible to exchange the communication infrastructure from TTP/C to TTE or any other protocol that suffices the claims stated in the DECOS project paper without affecting the layers above [6].

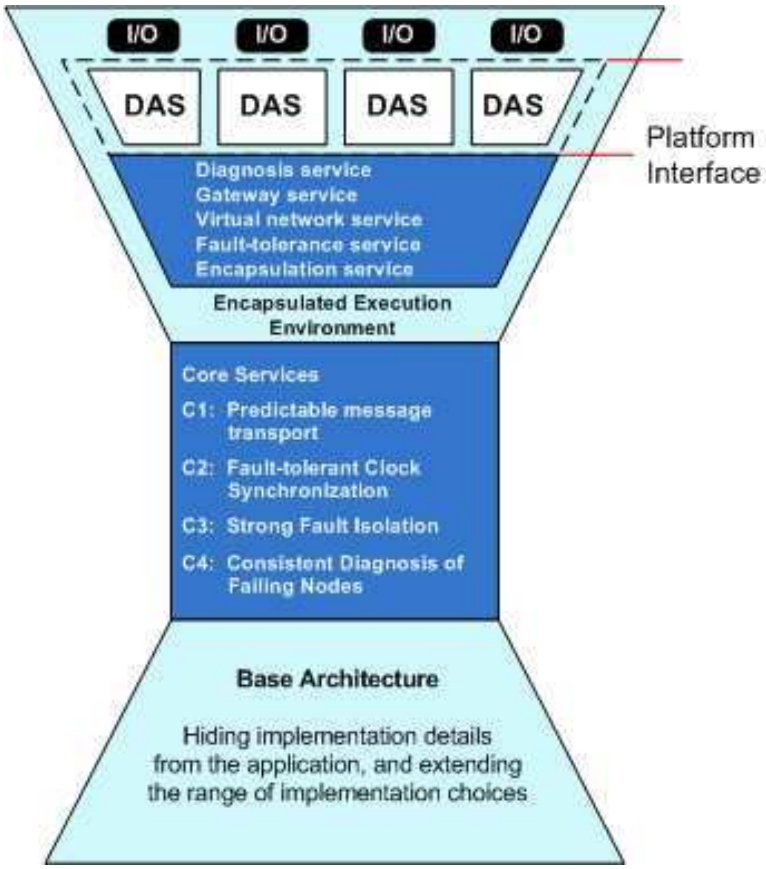

Figure 11: DECOS layered architecture [6]

The second layer contains the *DECOS core services C1 to C4*, therefore called *core services layer*, which get discussed later. These services are implemented

inside a unit called *communication controller (CC)* embodying the physical network adapters to the time triggered network.

On top of the core services, the *connector layer* provides the *DECOS high level services* like gateway services, virtual networks, encapsulated execution environment and diagnosis. This is the interface for the application developer and provides the abstracted interface to the cluster hardware.

A typically DECOS implementation in the automotive domain can be seen in Figure 12 on page 45, taken from [5].

3.1.2 DECOS core services

The core services [49, 6] are physically decoupled from the application hosts by using different computer hardware for the CC. Specified core services are categorized in classes C1-C4:

C1: Predictable message transport
Timely message passing is achieved by using a time triggered-protocol as the cluster communication medium. By implementing an a priori defined, free of conflicts TDMA bus access schedule, the temporal integrity is guaranteed. A temporal firewall between the application and the communication medium is thereby established [50, 2]. Faults inside a CC get detected by the bus guardian which guarantees that only temporally and semantically correct messages arrive at any CC in the cluster. Communication between the CC and the upper HL services is done by using message buffers in the CNI. No control messages cross the boundaries (except when using the message push mechanism). Message passing means transfer of messages from the CNI of one node to the CNI of another node in the cluster.

C2: Fault tolerant clock synchronization
Clock synchronization is performed for synchrony of the global time in the cluster. A *primary* and a *secondary rate master* send the synchronization messages on a periodically basis. DECOS uses a *sparse time base*, where messages can only occur at a given time, which makes it possible to synchronize at any given time a message arrives [2]. The CC detects whether the message arrived in the expected timing window based on the preconfigured schedule and either accepts or drops the message. The TTE rate master nodes are implemented at the level of the CC, meaning a CC of the cluster contains the primary rate master functionality and a second CC the secondary rate

master. No additional dedicated nodes need to be introduced to provide the clock synchronization service.

C3: Strong fault isolation

The CC performs basic syntax checking on the message format and error checking in the temporal domain on the messages received and to be sent to other CNIs. The CC builds an error containment region with the basic connector unit housing the high level services, which means faults in the HL-services (forming another fault containment region) causing incorrect data in the CNI and vice versa are at least detected in the communication controller [2].

C4: Consistent diagnosis of failing nodes

Information on the own current view on other connected nodes is done by tracing a bit pattern in the membership vector. The membership vector has an entry for each node in the system [2]. Information whether another node is alive or has failed can easily be obtained by monitoring the messages received from this particular node. As each node has the complete scheduling table for the TT messages, it can distinguish if a message was missing, did not arrive at the expected instant, received outside the reserved time slot or has an incorrect CRC value, and mark this node as "failed".

3.1.3 DECOS high level services

DECOS high level services are the interface to the application jobs [6, 2]. Depending on whether the DAS is a safety-critical or a non-safety-critical DAS, slightly different interfaces are exposed.

Virtual Network services

DASs exchange messages with other DASs over the network service. DECOS allows no direct access on the cluster communication bus due to encapsulation. The message passing among depending jobs is established by building *virtual networks* inside the time-triggered cluster message exchange [6]. This way the application jobs can be distributed among the cluster: No matter in which node the code is executed the virtual network connection is transparent to them. Virtual networks are invisible to unconnected nodes, which means they have a different namespace, preventing interferences due to duplicate names. The integration of additional applications into a cluster is therefore much simplified, as no changes have to be made to existing application namings. As information from virtual networks gets transported

over the TT network, the information has to be passed over to the communication controller. This is established by allocating a buffer for the size of the message inside the CNI and registering the message in the TDMA schedule.

DECOS provides two different classes of virtual networks:

- Time triggered virtual networks (TT VN)
- Event triggered virtual networks (ET VN)

Time-triggered virtual networks are used for safety critical application jobs and transport state messages [2]. When reading state information, which is actually the data in the CNI buffer, the application always gets the last value (i.e., no queuing is implemented). TT virtual networks inherit the properties from the time triggered protocol.

Event triggered virtual networks, the second class of networks in DECOS, allow the transmission of event information to other application jobs. Information sent from a job to the connected jobs of the respective ET VN is queued inside their CNIs. The communication uses a *producer-consumer* paradigm, which means the receiver-jobs have to actively remove the information from their CNI buffer by reading the message to prevent a buffer overflow after several consecutive transmissions.

Gateway services

Gateways in DECOS are responsible for the connection of two different DAS [2, 49]. We can distinguish between 2 classes of gateways:

- Virtual gateways
- Physical gateways

Virtual gateways connect two different virtual networks inside a cluster. There is no restriction on the type of the connected virtual networks - either VN of the same type can be interconnected, or TT and ET networks can be coupled, where the virtual gateway handles the temporal aspects to keep the temporal integrity of the TT virtual network.

Physical gateways can attach networks of different physics, like TT virtual networks and field-buses (e.g. CAN). The gateway service transforms the messages into the required format, keeps the timing constraints and also prevents error propagation among the connected buses.

Fault tolerant service

The fault tolerant service allows the establishment of redundant application jobs for safety-critical applications [2, 49]. To establish *Triple Modular Redundancy (TMR)*, three jobs of the same functionality can be grouped, and a *voting mechanism* ensures the correct service in case of a failing job.

Diagnostic service

Diagnostic services can perform basic checks on the job input and output, and give information on failing jobs [2, 49]. The diagnosis code of jobs has to be implemented in close cooperation with the job implementation. Diagnosis messages are transmitted via ET messages on the underlying communication channel.

Encapsulation service

The Encapsulation service, realized by an *Encapsulated Execution Environment (EEE)*, is responsible for the proper temporal and spatial partitioning [34]. It shall assure that jobs running on the same node do not affect each other. A fault in one jobs shall not cause interferences on other jobs.

If a TT job does not finish in its defined time slot, the *partition window time*, the job shall be terminated and reported via diagnosis service. An ET job does not have this constraint to finish its execution in the time slot. ET jobs not finished execution at the end of the configured partition window time shall be interrupted. On the next activation of the ET job, the job shall continue to operate at the point it was suspended.

Furthermore access to memory and buffers of jobs must be protected from the access from other jobs [34, 5].

3.1.4 DECOS fault hypothesis

A fault hypothesis contains specified *fault containment regions (FCR)* the *type of faults*, the *rate* of occurrences and *how* components will fail [36]. For the DECOS integrated architecture the fault hypothesis consists of a separate hypothesis for hardware faults and for software faults. The following is a summary of the fault hypothesis from [35].

Hardware fault hypothesis

- Fault containment regions

1. Node computer: Each node computer forms a FCR. A single failure will affect the whole node computer. Hardware diversity is used for redundant units.
2. Communication channel: Each communication channel forms a FCR. For replicated channels, each channel forms a distinct FCR.

- Failure modes
  - Node computer: Faults in node computers are arbitrary.
  - Communication channel: No spontaneously created correct frames and no arbitrary delays will occur.
- Failure rates
  - Permanent hardware failures: 100 *Failures in Thousands of million hours (FIT)* [37] which corresponds to at about 1000 years between two consecutive failures
  - Transient hardware failures: 10.000 - 100.000 FIT
- Maximum number of failures
  Single Fault Hypothesis: A single hardware failure of a FCR is assumed. After the recovery from a transient failure, another permanent or transient failure can be tolerated, whereas permanent failures require maintenance.

Software fault hypothesis

A distinction is made upon *system software* (middleware services, partition services) and *application software* (DECOS jobs).

- Fault containment regions
  - For software faults in the *system software*, the whole node computer forms a FCR, as a fault in this component will most likely affect the whole node.
  - For faults in the *application software*, each job forms a separate FCR.
- Failure modes
  - Communication system:
    An *arbitrary timing failure* is a failure where a message is sent at an instant not defined in the interface specification on timing

constraints. An *arbitrary value failure* is a failure where a message value is not defined in the interface specification on value constraints.

– Execution environment:
An *arbitrary timing failure* is a failure where an application job is violating the deadline or executed at a non-specified instant. An *arbitrary value failure* is a failure where an application job tries to access a memory region outside of the own partition or tries to access a port not specified for the job.

- Failure rates: according to *Safety Integrity Level (SIL)* levels 1,2,3,4. SIL4: probability of failure per hour: $\geq 10^{-9}$ to $< 10^{-8}$ [41]
- Maximum Number of failures according to SIL levels 1,2,3,4 [41], see above

3.1.5 DECOS cluster implementation

The DECOS architecture laid out in the paper [5] specifies the core services the framework shall provide and is not bound to the actual implementation used for the validation process in this paper. Using a Linux-based set-up gives the developer a high degree of flexibility in the development and debugging process.

The OS kernel is compiled to give full access to the network mechanisms like TCPIP networking, BSD like network sockets, mounting network shares over NFS which is used to receive the application binaries, and serial console output, used in the validation process to receive feedback about the executed experiments.

In contrast to a VHDL centric design, it is much easier to reconfigure the system and reload the complete cluster with a completely different application. As all nodes receive their network set-up, scheduling parameters and applications (the *workload*) from a network file server (DECOS server), it is a simple task to share the cluster among different developer groups. To switch from Application A to Application B it just requires to set the symbolic link for the main application download location from development group Directory A to Directory B. After restart of the cluster, the node computers fetch the new applications from the network share accordingly.

DECOS cluster hardware

All participating network nodes are implemented by Soekris microcomputer boards [5]. The boards are i386 compatible systems using AMD Geode processors with 64 MB RAM, a compact Flash slot for the storage medium with the initial Linux kernel, and at least two ethernet network interface cards (NIC).

According to their hardware requirements, the BCU/CC units are implemented on Soekris net4802 boards, having two onboard NIC's and an additional onboard mini-pci interface with three additional ethernet cards, so a total of five network interface cards exists. The ethernet network interface cards are used for the two TTE connections to the TTE switches, the two connections to SCU and XCU node computers and the connection for the diagnosis service.
The SCU and XCU node computers are implemented on separate Soekris net4521 boards, having two onboard ethernet NIC's for the connection to the diagnosis service and an additional PCMCIA NIC for the connection to the BCU/CC node computer.

3.1.6 DECOS scheduling

DECOS applications jobs in the current implementation of the Technical University of Vienna are RTAI-LXRT tasks [5]. This ensures the claims on timeliness in Linux operating systems. The scheduling of the three jobs per SCU/XCU is realized by an extra RTAI task, the DECOS scheduler, implementing a fixed scheduling strategy according to a design-time configured schedule table. This DECOS scheduler acts as the trigger for all DECOS application jobs. The trigger for the scheduler is a *magic packet* sent from the BCU/CC unit to the SCU/XCU units.

Scheduling in the DECOS cluster of the TU-Wien is realized by a kind of two-phase scheduling mechanism. The node scheduler gets triggered from the clusters BCU/CC node by sending a magic packet on the real-time network interface card. This interrupt gets intercepted by the RTAI layer and is routed to the RTAI task *scheduler*. The scheduler in term gets resumed and triggers the application jobs w.r.t. the design-time configuration and suspends itself for the time the job is allowed to run. After the defined partition time has elapsed, the scheduler gets reactivated by an RTAI timer function. Due to the higher priority of the scheduler task, the scheduler is in command of the CPU even if an application job has not finished its execution yet. Immediately after activation of the scheduler, it forces the application

job of the current partition time to suspend and activates the next application job which is defined for scheduling. After the last job of the TDMA round, the scheduler passes over control to the Linux operating system for a configured amount of time (`LINUX_TIME`). The scheduler is activated after this `LINUX_TIME` and blocks itself until the next activation by the network interrupt.

3.1.7 Example of a DECOS implementation

Figure 12 shows a typical DECOS implementation, handling the functionality of a car [5]. The car functionality comprises of a set of safety-critical (Drive by wire) and non safety-critical (navigation, lights, comfort) DASes. Safety-critical and non safety-critical jobs of the DASes are implemented side-by-side on the same component (DECOS node). Actuators and sensors with CAN or LIN interfaces can be attached via gateways in the DECOS nodes.

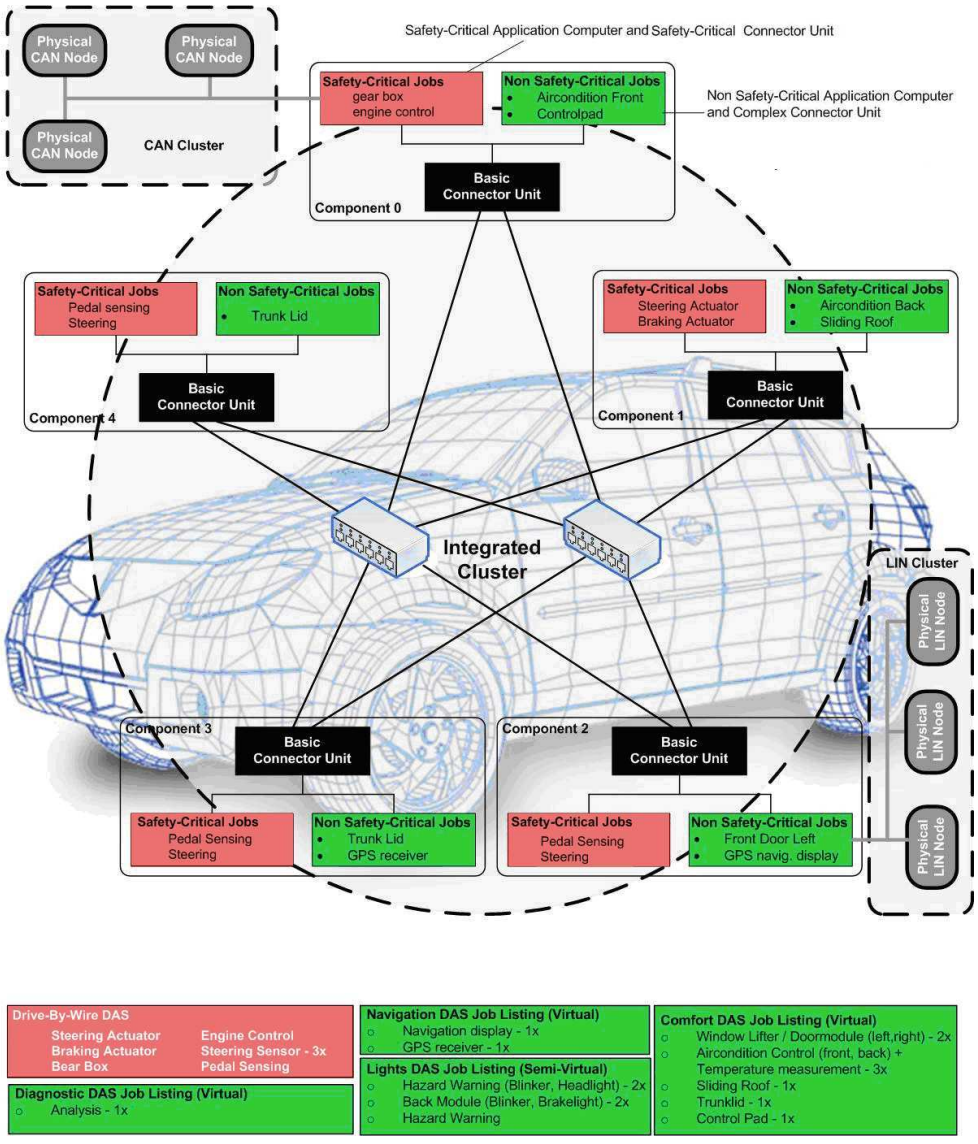

Figure 12: Sample DECOS infrastructure [5]

3.2 TTE - Time Triggered Ethernet

The implementation used throughout this thesis makes use of the Time Triggered Ethernet (TTE) protocol, so the following chapters shall give a brief overview about the services this protocol provides.

3.2.1 Aspects of TTE

The Time Triggered Ethernet makes use of the well established Ethernet protocol and infrastructure to guarantee deterministic message delivery among connected nodes [3]. The basic architecture is depicted in Figure 13.

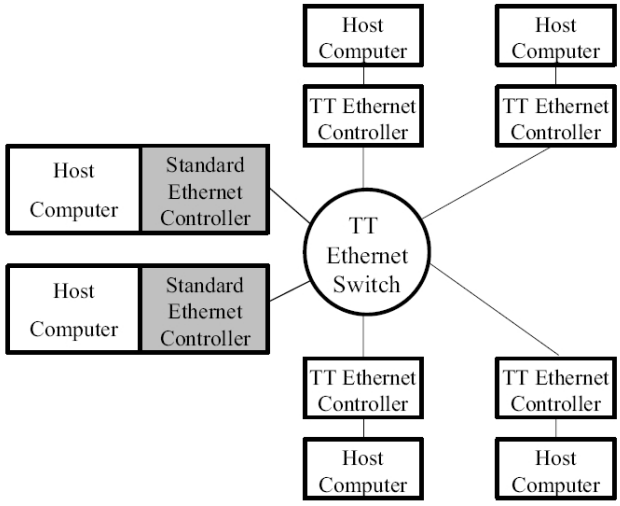

Figure 13: Time-Triggered Ethernet infrastructure [3]

TTE is a set of services and a dedicated TTE switching hardware to prioritize and control the medium access of the connected nodes. This approach does not limit the standard Ethernet mechanism in any way. In a TTE network standard desktop workstations can coexist with deterministic TTE nodes and will not be limited in their network functionality in any way [3]. It is one of the main advantages of TTE in contrast to other approaches reusing existent Ethernet infrastructures, where standard Ethernet traffic is not allowed because it will break the aspired determinism.

TTE defines a set of two categories of network packets in order to achieve the prioritization [3]:

- Event triggered (ET) messages

- Time triggered (TT) messages

Event triggered (ET) messages incorporate the standard ethernet messages and thus allow the existence of non real-time nodes in the network segment with services like *FTP, SMB, NFS*.

Time triggered (TT) messages are used by real-time network nodes to establish a deterministic communication. The message handling for TT messages is presented in Section 3.2.2. Time-triggered messages have a specific identifier in the ethernet *frame type*-field, which was defined by the IEEE organization as identifier $0x88d7$, to allow a distinguishment from standard ethernet messages.

The whole infrastructure of the communication medium, the TTE switch and connected nodes, whether real-time or non-real-time, is called the *TTE cluster* [3].

3.2.2 TTE switch

The TTE switch is the core component in the TTE cluster communication, as it is responsible to guarantee the timely delivery of these *real-time network packets* among the participating nodes. Standard Ethernet switches are implemented in a way that packets are delivered in the order as they arrive - the *First in First out* mechanism. Opposing to that, TTE switch implementations detect real-time packets by their frame type field value of $0x88d7$. Once the switch detects such message, currently transmitted ET messages get *preempted* to free the communication channel and get stored in the internal message buffer for subsequent retransmission [3]. The delay for TT messages is thus a *a priori known constant factor* — the time necessary for clearing the transmission path — and depends on the actual implementation of the TTE switch.

3.2.3 TTE nodes

Real-time applications in the TT Ethernet Protocol are handled by TTE nodes. According to [3], the TT Ethernet protocol defines four layers for the implementation of a node. Proceeding top down, these are:

Host computer
This layer incorporated the application job(s) of the node and processes the workload in the system. The applications can access the abstracted hardware

and communicate with other nodes over a fixed API set described in [3]. Messages for sending to other nodes get written into the *Communication network interface (CNI)* buffer by using the TTE API functions. Messages of interest to the application also get read from the CNI's buffer. The host computer has no direct interface to the underlying network hardware [3, 51].

Hardware abstraction layer (HAL)

The hardware implementation of TTE nodes is not bound to specific architectures [51]. To hide those design varieties from the application developer, the HAL provides a common interface to the application layer among all possible hardware design variants and keeps the applications portable.

Communication network interface (CNI)

This layer is responsible for the data exchange between the host computer and the TT Ethernet controller. The CNI keeps the configuration data for all messages to be processed by the host computer. That are the message size, the access type (push or pull) and the timing information. According to this information the CNI allocates the required memory to buffer all message types [51]. All required configuration data is defined before the compilation. However, the configuration can be dynamically updated during the runtime. Messages received either from the application tasks of the own node or from the network are written to the reserved memory. A received message from the TTE controller is updated in the buffer and checked for follow-up procedures: If an application has defined *message push* for this message type, the CNI generates an interrupt for the application.

Time Triggered Ethernet Controller

The Time Triggered Ethernet Controller is the interface to the communication medium and ensures the correct reception and timely transmission of messages to and from the cluster. According to the configuration file stored inside the CNI, the TTE sends the messages from the CNI buffer at the correct instant of time [3, 51]. All messages types in the configuration file must be defined as such that no two or more messages have he same instant for sending (i.e., it must be ensured that the sending schedule is free of conflicts). This is vital for the correct function of the TT protocol, as the TTE switch does no buffering of TT messages, which would violate the timeliness of the messaging subsystem.

3.2.4 Global time

Elementary for the claim of timeliness in the whole TTE cluster is the installation of a common and tight synchronized time base, the global time [3]. All nodes must have a common understanding of the current instant, so that the TT Ethernet controllers can deliver the network messages free of collisions in the reserved time slot. The global time format of the TT Ethernet protocol is expressed in the Universal Time format as shown in the following picture:

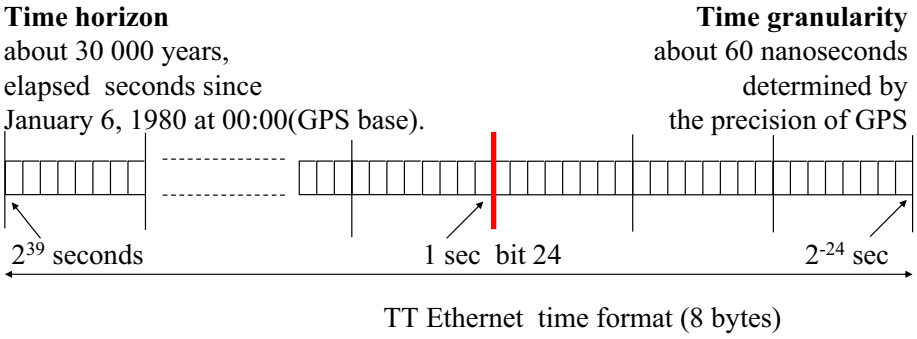

Figure 14: TTE time format [5, 51]

The *granularity* of the TTE time — the second fractions — are expressed by a 24 bits wide value and the horizon is presented in a 40 bit field which gives a total length of 8 bytes. Using a *horizon* of 40 bits gives a presentable time range of over 30.000 years, which is far enough in the future for current applications. 24 bits for the granularity in term means

$$\frac{1}{2^{24}}\ seconds = 59.6\ ns$$

A tick of the global time is called a *macrotick*. To express the dependency from the cluster time ticks to local time ticks, the factor *microtick macrotick conversion factor (MMCF)* is introduced. A MMCF of 100 means that 100 ticks of the local clock correspond to a single tick of the global time.

3.2.5 Clock synchronization

As we have learnt from the chapter on real-time operating systems, crystal oscillators are no *perfect clocks* due to their drift. To ensure synchronized clocks among all nodes in the TTE cluster, two special TTE nodes are introduced, namely the *primary rate master* and the *secondary rate master*. The primary rate master is the *leading clock source* within the whole cluster.

All other nodes, called *time keeping nodes*, correct their clocks with respect to this node as long as the *correction value* is within a well defined range, the maximal external clock correction factor. To accomplish this, special TT messages are sent to all participating nodes, the *TT synchronization messages*. The secondary rate master acts as a hot stand-by in case the primary rate master fails. Synchronization messages get sent by both of the rate masters, but at different instants. During normal running conditions, the secondary rate master node also synchronizes its own clock with respect to the TT sync messages of the primary rate master [3]. It is evident that the clock generator of the primary and the secondary rate master should have the highest precision in the system.

3.2.6 Safety critical TTE

Standard TTE relies on the fact that the transmission schedule of messages is free of conflicts in the temporal domain, nodes are free of faults during the whole runtime and the TTE switch is free of defects during the runtime. For safety critical applications, one can not rely on these facts but has to introduce special measures to ensure the correct operation of the system even in failure cases, at least with possibly degraded system performance. TTE was already designed with safety-critical aspects in mind, so the required modifications to upgrade existing applications can be classified as minor. Most of the safety critical extensions are hidden from the host computers handling the cluster workload [3, 51].

A look at a typical safety-critical TTE cluster infrastructure shows the distinct fault containment regions:

Failure inside a TTE node

Nodes, as we can see in Figure 15 are the components embodying the host computer and the TTE controller. Failing application jobs can therefore be one source of a failure. We already discussed critical functions in RTOS like dynamic memory allocations, dynamic task creation, or infinite loops. These issues are handled in the chapter of validation of the EEE later on. The second subsystem in the node is the TTE controller. A failure in the TTE controller, either timing failure or value failure, can be detected from the bus guardian of the TTE switch.

Failure inside TTE switch

A failure in the TTE switch can either be failing hardware of the switch or a failure in the bus guardian application. From the point of view of the cluster

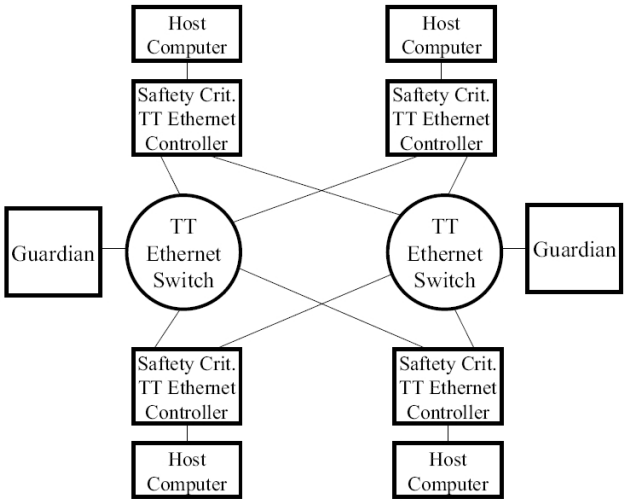

Figure 15: Safety critical TTE [3]

nodes, a failure in the TTE switch can not always be clearly distinguished from a problem with the communication channel. To tolerate failures inside a TTE switch, a redundant communication medium is required.

Failure of communication channel

A failure of the communication channel can be a broken connection. To tolerate such failure, redundant communication channels are required.

TTE is designed so that faults not covered by the fault hypothesis at least get detected and reported (*never give up strategy*) [3].

3.2.7 Enhanced SC TTE switch - Bus guardian

The TTE switch implementation from the basic TTE infrastructure (Figure 13) has no mechanisms to detect arbitrary TT messages sent from failing nodes. A corrupt node flooding the TTE switch with temporal invalid TT messages — called *babbling idiot* [1] — can affect the whole system. The basic TTE just prioritizes TT messages over ET messages and handles TT messages in the order as they arrive. In the event of a failure, where two TT messages arrive at the same instant, it depends on the actual switch implementation which message gets transferred onto the communication medium and which one gets dropped, and is therefore lost [51].

This is not acceptable in safety-critical environment and can only be prevented by an intelligent bus arbiter, a *bus guardian* inside the TTE switch (Figure 15). The bus guardian has basic knowledge about the a priori defined transmission schedule, the TDMA schedule, and can therefore decide, whether a message received from a node is valid or not. Messages from connected nodes received outside the defined time window can positively be identified and refrained from affecting the cluster communication [3]. This way a *babbling idiot* can be isolated from the cluster. The bus guardian listens to the TT messages and can perform basic analysis on the TT messages. In case that an incorrect message was observed, the guardian can disable the TTE switch port of the affected, possibly failed node and disconnect it from the network. Additionally the guardian can check, if the TTE switch itself behaves according to the timing specifics like the constant latency that is introduced by the TT message forwarding mechanism.

4 The SWIFI Framework

This section describes the SWIFI framework for the emulation of the software faults required for the validation of the EEE. The SWIFI framework for the validation of the DECOS integrated architecture shall allow the emulation of software faults specified in the fault hypothesis of the DECOS integrated architecture (see Section 3.1.4). The SWIFI framework described in this section is an extension to an already existing framework [15], which allows the injection of hardware faults into DECOS nodes.

4.1 DECOS infrastructure

The test setup infrastructure of the DECOS cluster implemented at the Technical University of Vienna contains five DECOS nodes, each consisting of the units BCU/CC, SCU, and XCU. The Time-Triggered Ethernet (TTE) protocol is used as the core communication architecture. The schedule of the TTE contains two TDMA rounds, each round 16 ms long. Each of the five slots of a single TDMA has a duration of 3.2 ms. Each slot allows an independent schedule of the three application jobs, denoted as jobs in the DECOS terminology.

The Encapsulated Execution Environment (EEE) is run on the safety critical connector units $SCU[1-5]$ and on the complex connector units $XCU[1-5]$. Its purpose is to allow the execution of different application jobs on the hardware units SCU and XCU. Furthermore, the EEE is responsible for the resource sharing, according to the user specified configuration, and encapsulation of the executed jobs. The transmission of messages is done according to the fixed static schedule configured at design-time. On the occurrence of faults inside the EEE, the mechanisms shall handle the faults specified in the DECOS fault hypotheses.

Time-Triggered (TT) jobs shall be controlled in a way that no TT job exceeds its defined *partition window time*. If a fault inside a TT job, which causes the job not to finish its execution within the partition window time, the EEE shall terminate the faulty job and trigger the diagnosis service to record the deadline violation. Should an ET job take longer to finish, the EEE must not terminate the ET job at the end of the partition window time (see Section 3.1.3), but suspend it.

The workload processed by the DECOS cluster is a *Brake-By-Wire (BBW)* application. The BBW application is a *Distributed Application System (DAS)*,

distributed over five node computers. A distinct *wheel function* module for each of the four wheels of a car, a *brake pedal function* and a *main function* form the DAS.

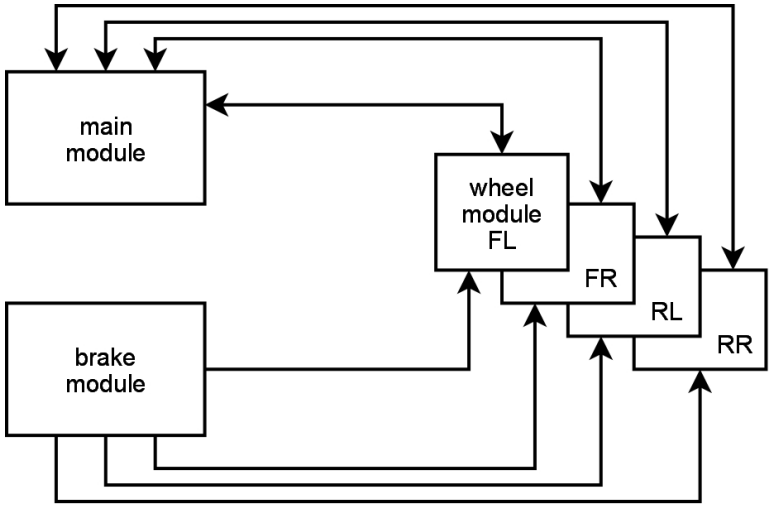

Figure 16: BBW DAS structure

The main module calculates the vehicle speed upon the information of the four wheel modules. The brake force applied on the brake pedal is emulated by the *brake job* in the *brake module*. Each wheel module calculates the braking force to be applied on its wheel's brake based on information from the

- vehicle speed,
- wheel speed, and the
- brake force applied on the brake pedal.

As the BBW system is considered as safety-critical, the *wheel function* is replicated. Each wheel node houses the functionality of two other wheels as *shadow units*.

The system under test is a DECOS cluster, consisting of five DECOS nodes whereas each node is composed of a Safety-critical Connector Unit (SCU) and a compleX Connector Unit (XCU), implemented on Soekris net4521 computers, and the Basic Connector Unit (BCU) and Communication Controller (CC), integrated in a single Soekris net4521 computer.

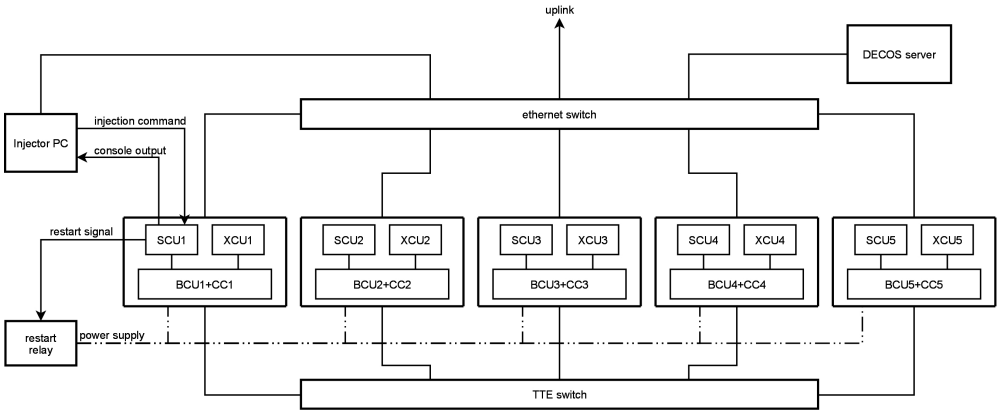

Figure 17: System under test

In the five DECOS nodes, each SCU and XCU unit has an ethernet connection to the respective BCU/CC unit. All BCU/CC units are connected via a separate ethernet connection to the TTE switch. This connection handles the time-triggered ethernet communication for the exchange of the messages of the core services and the virtual networks.

For diagnosis purposes, the loading of application code into the SCU and XCU units, and transmission of the fault injection commands, each unit (SCU, XCU and BCU/CC) has an additional ethernet port connected to a separate ethernet switch. This switch also connects the DECOS server and the fault injector PC to the DECOS units. The DECOS server provides a network share, where the application code of the SCU and XCU units is supplied. On startup of the DECOS cluster, the SCU and XCU units fetch their application from this share and start the execution of the application jobs.

The initial DECOS cluster had distinct node computers for the BCU and CC units. During development of the extended framework and experiment application jobs, the cluster had undergone system changes, where the BCU and CC were merged into a single device. Table 1 shows the cluster's IP addresses of the diagnosis ports.
For the injection of faults according the predefined fault cases, either $SCU[1-5]$ or $XCU[1-5]$ acts as the *fault injector unit* in the DECOS cluster. This unit receives the injection command from a standard desktop PC running a Linux operating system.

index	0	1	2	3	4
	node-IP: 192.168.0.				
BCU+CC	100	104	108	112	116
SCU	102	106	110	114	118
XCU	103	107	111	115	119

Table 1: DECOS-cluster IP configuration

An additional unit in the cluster - SCU_0 during the experiments - acts as a *reset unit*. The reset unit can interrupt the power supply of the whole DECOS cluster hardware to restart the system by switching a relay. This is necessary to put the system in a consistent state after a fault injection experiment is executed.

4.2 SWIFI software infrastructure

Figure 18 shows the architecture of the involved software components in the SWIFI framework.

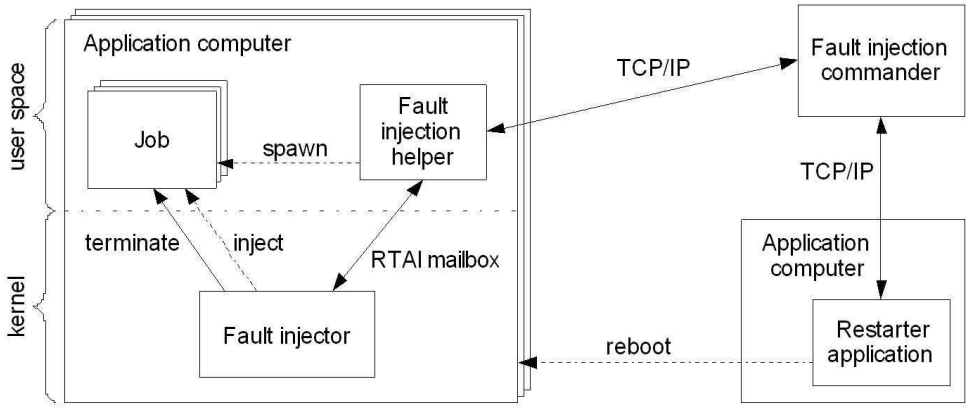

Figure 18: SWIFI infrastructure [15]

According to the specification [15], the software infrastructure contains the following software components:

Fault Injection Commander (FIC)

The fault injection commander is the controller of all experiments. For the experiments executed in this thesis, the fault injection commander is run on a standard desktop PC with a Linux operating system. Experiments to

be run by the FI commander are defined in a standard text file described in Section 4.4 and [15]:

Fault Injection Helper (FIH)

The fault injection helper is an application task running on the unit under test, the *fault injection unit* in the DECOS cluster, which can either be a *SCU* or *XCU*. Its purpose is to communicate with the fault injection commander and to distribute the commands in the DECOS cluster. The fault injection helper runs in the user space, as it has to communicate over TCP/IP with the fault injection commander, which is not trivial to perform in kernel space [15]. As soon as the *fault injection helper* receives a command from the FIC, the command is passed over to the defined fault injector subsystem, which has to be executed in each unit where an experiment is scheduled. The unit *SCU* or *XCU*, where an experiment shall be executed has to be defined in a configuration file at compile time of the DECOS application in order to include the *fault injector* and *fault injection helper*. These two components are transferred to the unit along with the application jobs via download from the DECOS server at startup of the cluster.

To perform the actual experiment, the FIH analyzes the command received from the FIC over the network and passes it over to the fault injection helper of the respective unit. The communication between FIH and *fault injector* is done by the RTAI primitive `mailbox`. The transfer by RTAI mailbox is done by the function *rt_mbx_receive_wp()*, which allows a non-blocking message passing. The call of the function allows the reception of a message without blocking the calling task [20].

Fault Injector

From Figure 17 we can see that the fault injector receives the experiment command from the fault injection helper by a *RTAI mailbox-function*. According to the intended injection it waits for the defined time instant when the injection shall be executed and manipulates the addressed memory or shared memory accordingly [15]. The fault injector also performs the calculation of the memory address where the injection shall be executed. It is also possible to terminate the job after the experiment. The fault injector may terminate the job execution after a specified timeout expires, relative to the point in time where the fault injection restart command is received. For most test cases this was not used, as the monitoring of the results was partially observed on the serial console output of the DECOS *SCU* or *XCU* units. It was observed that the reaction on an injection command of the experiments

in several cases was observed even after the timeout time. A reset before the result can be observed would render the experiment run useless. It was also helpful to get some information after a crash of an application job via console, like the memory usage, or the number and names of RTAI objects instantiated when analyzing particular experiments.

Restarter Application

The restarter application shall be executed on a single *SCU* or *XCU* of the DECOS node. Its purpose is to execute a reset command after the specified consecutive experiment commands in order to bring the cluster in a defined state [15]. In the current implementation, it is possible to issue six commands per experiment, although this can be extended. The restarter application uses a single pin of the I/O interface to switch a relay, which interrupts the power supply of the cluster. This can be used to run a batch of experiments, read the results back and issue the reset of the application computer before the next experiment run is started. To observe the results of an experiment crashing the node computer, the reset command was not used.

The original framework mentioned described in [15] provides support for memory faults in application jobs and global memory. Although adequate for evaluating tests which shall emulate hardware faults inside of DECOS nodes like *bit-flip faults*, it lacks the support for the fault scenarios mentioned in the fault hypothesis on software faults. As the goal of this thesis is to test the functionality of the encapsulated execution environment for correctness in the event of software faults, it was necessary to define a set of test cases and implement this *incorrect code* in a DECOS job. To selectively *activate* the intended fault set, the original fault trigger mechanism of the existing framework was used.

The intended faults being injected like task deadline violation and illegal memory access are dedicated software application faults. All faults are in the scope of the the application jobs and no manipulation of the task's code segment, data segment or stack had to be triggered by the DECOS system or the fault injection layer of the FI framework. The experiments could comfortably be triggered by reusing the service to write into shared memory areas of dedicated nodes.

4.3 Fault Injection Command

To start the experiments on specific units, the fault injection commander has to send several parameters to the involved units. The basic approach from

the existing framework to write a bit or a byte to an absolute location was not sufficient for the experiment parameter transfer. The FI commander has no knowledge of the absolute position of the control variable which in term distinguishes among the intended experiment classes.

Although it would have been possible to get the memory address of the experiment selection variable in the initialization sequence of the specific unit and read that address when executing the experiment, it would introduce an additional delay and require an enhanced fault injection commander for generating dynamic test commands. The sequence for a dynamic generated test could be:

1. *SCU* or *XCU*, init-function of job: get absolute address of variable "var_x" and write it into shared memory "SHM_x"
2. FI commander, experiment start: read value of shared memory "SHM_x"
3. FI commander, experiment start: form a command for "absolute address injection" by specifying the address as read value from "SHM_x"

Another option is to write to the shared memory provided by the RTAI subsystem. This mechanism was used for the implementation of the extended SWIFI framework. To manipulate the shared memory by the FI system one can use a command in the form like:

```
SCU0, shm=FITEST, 3, ---, byteset, 1, ---, concrete, 0
```

This command writes the value "1" to the shared memory named "FITEST" on the safety-critical unit 0 (SCU_0) in the third TDMA slot. This shared memory location can be mapped by using the RTAI-functions into the affected application jobs. The application jobs can then react on invocation by the scheduler on the value present in the shared memory address and trigger the experiment run accordingly. The framework allows to transmit byte values only. This means, a maximum value of 255 can be sent by the fault injection commander via a command to the fault injector. For values greater than 255, the value has to be split into its byte values, and the command has to consist of several consecutive writes to different shared memory regions for all required byte values. This limitation is by design of the fault injection framework.

The transmission via shared memory is used either for signaling the type of experiment to be run, and also for additional parameters, required to

accomplish the experiment run. For example, tests of the temporal encapsulation require an parameter for the varying delay time the job shall wait before finishing execution.

4.4 SWIFI Framework capabilities

Original framework

The original framework was implemented for the injection of hardware faults into a DECOS node computer as specified in [15]. An experiment is started by running the fault injection commander which reads the commands from an *experiment configuration file (ECF)*. The syntax of the ECF is presented below. An experiment can include a maximum of six injection commands. Each injection command is defined in a separate text line of the ECF containing the full description about timing, destination and value.

The first parameter defines the DECOS unit where the injection shall be executed. The framework allows to specify the targeted unit by the unit name ranging from *SCU[1..5]* and *XCU[1..5]*.

The destination of an injection can be defined in the second parameter. Supported destinations are:

- memory of an application job (*job[n]*)
- absolute memory address in the unit's memory (*global*)
- RTAI shared memory (*shm =<RTAI object name>*)
- memory of a kernel module (*module =<module-name>*)

Parameter three defines the injection timing. It's value presents the time slot in the communication schedule, when the injection shall occur. For a finer granularity of the injection instant, the fourth parameter denotes a timing offset relative to the task activation. The delay can be specified in nanoseconds. If an offset value $t > 0$ is given, the fault injector is started t *ns* after task activation. If a time *t=0* is given, the injection is performed prior to task activation.

The fifth parameter of an injection command is the fault type, which can either be *bitflip* or *byteset*. The "bitflip" fault is a common fault model in the aerospace domain, where due to the hit by an energetic particle a false signal in an electronic circuit is generated, which causes a memory cell to changes it's value. The "byteset" injection is used to emulate a memory

fault, where a memory cell gets defective (stuck-at-0, stuck-at-1).

Parameter six defines the data to inject. If the fault type is "bitflip", a value in the range [0..7] defines the bit position to flip, whereas a value of -1 is a random bit position. For the fault type "byteset", a value in the range of *[0x0..0xFF]* defines the byte value to set, and -1 means a random byte value.

If a fault injection is to be performed into an application job's memory, parameter seven specifies whether the memory area targeted shall be the

- data space of application job
- code space of application job
- stack of application job or an
- absolute address of the application job

Parameter 8, 9 and 10 denote the address type (*concrete* or *random* address) and the memory range (*start* and *end* address).

Extended framework

The extension of the SWIFI allows the injection of software faults. The fault types to be injected are derived from the DECOS fault hypothesis. To execute the experiments of the fault hypotheses, the DECOS application jobs of the unit under test have to be modified to include the fault injection code. The fault injection code is implemented as part of a DECOS job of the SCU or XCU units.

For the experiments in this thesis, the application job job_2 was altered to receive the experiment command and the parameters required. The parameters are transferred by the mechanism of the original framework to write into a *shared memory* of a DECOS unit SCU or XCU.

The experiment parameters are written by the fault injector in the respective time slot as specified in the fault injection commander configuration file. This allows the triggering of experiments on a definite schedule. A separate DECOS job (*monitoring job*) is responsible for the feedback on the experiment runs via console output. A serial port of the Soekris node computer of the unit under test is used for serial console output. According to the specified experiment, the output on the serial console gives feedback whether the experiment run was successful or did not finish (i.e., crash).

4.4.1 Temporal encapsulation

The framework provides two different classes of experiments in the temporal domain (i.e., deadline violation of tasks):

Deadline violation of job by delay
In this experiment class, a task receives a parameter specifying the delay time for execution before finishing. The actual delay is achieved by performing a *busy_wait* function. The delay time of the *busy_wait* function was measured over several consecutive runs.

Deadline violation of job by infinite loop
This experiment causes an application job to never finish its execution cycle due to executing a infinite loop. Accordingly, the implementation uses a loop with a *busy_wait* function that will never finish.

As the injection code is in the application job, the experiments on deadline violation can be executed on either safety critical connector units (SCU) or complex connector units (XCU).

4.4.2 Spatial encapsulation

Illegal memory access by job
This experiment experiment class allows to attempt access to a memory segment of another job. A dedicated DECOS job specifies a variable address, whose value is periodically printed on the console output of the monitoring node. A second job tries to access this memory location by a pointer function, whose address is handed over via shared memory from one job to another job. On successful change of the contents of the memory location, the value can be observed from the monitoring job's console output.

Illegal comm port access by job
The fault injection task contains a code fragment, which tries to access the communication port of a job it is not specified for.

Stack overflow
This experiment covers the case, where an application job calls a recursive function, whose call depth is not limited. The experiment simulates a faulty recursive call, whose end can never be reached. This way a stack overflow is issued. The recursive function (see below) is called once each TDMA round.

On each invocation of the recursive function *recursivefunction()* a total of 1 kB is allocated.

```
long recursivefunction(long iteration){
   volatile long garbage[1024];
   garbage[1024-1]=rand();
   rt_printk("i: %d\n",iteration++);
   garbage[1024-2] = recursivefunction (iteration);
   return 0;
}
```

Dynamic memory allocation

This experiment contains test cases executing the RTAI-functions *rtai_malloc and rtai_kmalloc to allocate memory address in user-space (rtai_malloc*) and kernel space (*rtai_kmalloc*). The test shall prove the reliability of the EEE when performing the operations until no more memory is available for allocation.
Furthermore the ANSI-C functions *malloc*, *kmalloc* and *vmalloc* get tested. *kmalloc* allocates a contiguous memory area in kernel space, whereas *vmalloc* allocates a virtually contiguous memory area in kernel space. Virtually contiguous means, the RTAI-LXRT task sees the memory as contiguous, but the physical layout in RAM is not contiguous. The function *malloc* allocates the respective memory region in the heap of the task. This information can be read in Chapter 8 of the document "Understanding the Linux Virtual Memory Manager" [17]. For the experiment on all dynamic memory allocation functions, in every TDMA round an invocation of the respective function is executed.

Tampering the scheduling semaphore

The timing mechanism in the EEE is established by using a semaphore to signal the start of the core cycle to the EEE scheduler. After each timing slot, the scheduler executes the *sem_p()* function to block itself. The block is released on receiving a *magic packet* from the network driver. The network driver executes the *sem_v()* on the blocking semaphore and synchronizes the scheduler. This experiment shall test the effect when the *sem_v()* function is called from inside an DECOS job.

5 Experimental Results

This section presents the the experiments and results for the validation of the DECOS *Encapsulated Execution Environment (EEE)*. The executed experiments cover the faults specified in the fault hypothesis (see Section 3.1.4).

The Encapsulated Execution Environment (EEE) supports the execution of applications with different criticality on top of the same physical hardware. The mechanisms of the EEE shall establish the encapsulation of the resources in a physical node, with prior (static) assigned resources: CPU time, storage space (memory) and I/O access. The EEE shall provide mechanisms for the handling of software faults at the job level. The Encapsulated Execution Environment allocates statically the CPU time to different jobs in an application computer. The validation objective is to show that the allocated CPU time of different jobs in an application computer are guaranteed by the EEE even in the presence of the faulty jobs. The EEE allocates a time window called the *partition time window* for each job in an application computer. A given partition time window will be allocated periodically to a given job with a period denoted as core cycle. In the DECOS integrated architecture one job consists of one or more tasks. Time-triggered tasks are executed periodically and each task has to finish before a certain deadline is reached. All time-triggered tasks of one job have to finish within the given partition time window. In case that a task of a job has not finished the execution within the given partition time window the EEE shall interrupt the task and report on the deadline violation of the job. In case of event-triggered tasks, a task will be interrupted when the partition time window expires, but no error handling shall be performed because by definition event-triggered tasks are not supposed to have deadlines. The Encapsulated Execution Environment statically allocates memory and I/O resources for different jobs in an application computer and guarantees that no other job can access the memory resources allocated to one job.

5.1 Validation goals

A software fault in one partition should be isolated, and the process of isolation of a partition (task in that partition will be reset or restarted) should not affect the operation of other partitions. The validation of EEE is divided into two parts:

1. **Validation of the temporal encapsulation**
   The goal of the validation is to show that no software fault in one

job can affect the operation of the application computer such that the CPU resources (time slot) of other jobs are affected. The validation mechanism shall inject faults such that a time-triggered task misses its deadline. This can be achieved with the software implemented fault injection mechanism (SWIFI) that was developed in this thesis.

2. **Validation of the spatial encapsulation**
   The goal of the validation is to show that no software fault in one job can access the memory and I/O resources of different jobs, or can affect the operation of the application computer. Furthermore, it must be shown that a job does not access the input and output ports (used to communicate with the DECOS high-level services) of other jobs in an application computer. The validation mechanism will try to inject faults that cause a faulty job to access the memory and I/O that are allocated to other jobs.

5.2 Temporal encapsulation

> "Time triggered tasks are executed periodically and each task has to finish before a certain deadline is reached. In case that a task of a job has not finished the execution within the given *partition time window* the EEE (Encapsulated Execution environment) shall interrupt the task and report on the deadline violation of the job [8]."

5.2.1 Deadline Violation - violation by $x\%$ for TT jobs

Experiments are executed in SCU_0. The SCU_0 executes three jobs:

- job_1: emulation of the vehicle functions
- job_2: a dummy job that contains the code to do the actual fault injection. It gets the delay time value from the fault injection framework and issues the deadline violation, i.e. performs the fault injection experiment.
- job_3: the monitoring job to display the incrementing counter to observe the experiment results

In this test run job_2 shall try to execute longer than its allocated time interval, and thus miss its deadline. The fault injection experiments affect the execution of job_2 such that it causes various execution time of the job. The cluster is configured such that 320 μs are allocated for execution of each

job. The job job_2 is affected such that it will violate its deadline for $i\%$ of the amount of time allocated for the job (where i is chosen to be 10, 50, 90, 130). That means that the execution time of periodic part of this job is additionally extended for approximately 10% of 320 μs = 32 μs in the first experiment run up to 130% of 320 μs = 416 μs in the last experiment run.

Expected result According to the DECOS claims on time triggered jobs the EEE shall interrupt the faulty job and report the fault in the diagnosis service. The partition window given for job_2 is configured in the file "/config/scheduler_config.h". The fault injecting job job_2 is configured to run at the beginning of **every** MEDL slot (see C listing below) with a fixed *partition window size* of 320 μs which would mean the incrementing counter should get reset every 3.2 ms.

```
struct schedule_type task_schedule[MAX_MEDL_POS] = {
  { { /* MEDL_POS 0 */
      {2, 4*TIME_FACTOR},
      {0, 0*TIME_FACTOR},
      {0, 0*TIME_FACTOR},
      {0, 0*TIME_FACTOR},
      {0, 0*TIME_FACTOR},
      {0, 0*TIME_FACTOR}
  } },
  ...
  { { /* MEDL_POS 9 */
      {2, 4*TIME_FACTOR},
      {0, 0*TIME_FACTOR},
      {0, 0*TIME_FACTOR},
      {0, 0*TIME_FACTOR},
      {0, 0*TIME_FACTOR},
      {0, 0*TIME_FACTOR}
  } },
}
```

Experiment result

Note that every experiment run is executed at least three times. The results of the experiments are shown in Table 2.
The result table presents the experiment run ("run"), the delay introduced to the job ("delay"), the effect on the job executing the experiment, either terminated or running ("job_2 terminated") and the result of the experiment run, whether it was successful or not ("exp. passed"). Furthermore the states of the three jobs after the experiment runs are presented, whereas the state

can be either "run" if the job is still executing after the experiment, "stuck" in the event that a job is halted due to an error and stopped executing, or "released", if the RTAI-LXRT detected an error and terminated the job. The system state can be either "run" if this SCU or XCU continues to schedule the unaffected jobs, "stuck" if the scheduler of the SCU or XCU error was affected and stopped or "reboot", if the unit performed a reboot due to the fault injection.

run#	delay	job_2 terminated	exp. passed	Status job_1	job_2	job_3	system
1.1	10% (32 μs)	no	no	run	run	run	run
1.2	10% (32 μs)	no	no	run	run	run	run
1.3	10% (32 μs)	no	no	run	run	run	run
2.1	50% (160 μs)	no	no	run	run	run	run
2.2	50% (160 μs)	no	no	run	run	run	run
2.3	50% (160 μs)	no	no	run	run	run	run
3.1	90% (288 μs)	no	no	run	run	run	run
3.2	90% (288 μs)	no	no	run	run	run	run
3.3	90% (288 μs)	no	no	run	run	run	run
3.1	130% (416 μs)	no	no	run	run	run	run
3.2	130% (416 μs)	no	no	run	run	run	run
3.3	130% (416 μs)	no	no	run	run	run	run

Table 2: TT deadline violation (delayed)

5.2.2 Deadline Violation - violation by infinite loop for TT jobs

Experiments are executed in SCU_0. The SCU_0 executes three jobs:

- job_1: emulation of the vehicle functions
- job_2: a dummy job that contains the code to do the actual fault injection. It gets the command to enter the infinite loop from fault injection framework and issues the deadline violation, i.e. performs the fault injection experiment.
- job_3: the monitoring job to display of incrementing counter to observe the experiment results

Expected result According to the DECOS claims on time triggered jobs the EEE shall interrupt a faulty job and report the fault in the diagnosis

service. The operation of the application computer or the SCU/XCU shall not be affected.

Experiment result Note that every experiment run is executed at least three times. The results of experiments are shown in Table 3.

		job_2	exp.	Status			
run#	delay	interrupted	passed	job_1	job_2	job_3	system
1.1	∞	no	no	run	run	run	run
1.2	∞	no	no	run	run	run	run
1.3	∞	no	no	run	run	run	run

Table 3: TT deadline violation (infinite loop)

5.2.3 Execution time of an ET job longer than configured

> "In case of event-triggered tasks, a task will be interrupted when the *partition time window* expires, but no error handling shall be performed because by definition event-triggered tasks are not supposed to have deadlines [8]."

Experiments are executed in XCU_0. The XCU_0 executes three jobs:

- job_1: emulation of the vehicle functions
- job_2: a dummy job that contains the code to do the actual fault injection. It gets the delaytime-value from fault injection framework and issues the deadline violation, i.e. performs the fault injection experiment.
- job_3: the monitoring job to display of incrementing counter to observe the experiment results

In this test run, job_2 shall try to execute longer than it's allocated time interval. The fault injection experiments affect the execution of job two such it causes various execution time of the job. The cluster is configured such that 320 μs are allocated for execution of each job. The job_2 is affected such that it will execute longer than the allocated time by a factor of $i\%$ (where i is chosen to be 10, 50, 90, 130). That means that the execution time of

periodic part of this job is additionally extended for approximately 10% of 320 μs = 32 μs in the first experiment run up to 130% of 320 μs = 416 μs in the last experiment run.

run#	delay	job_2 interrupted	exp. passed	Status job_1	job_2	job_3	system
1.1	10% (32 μs)	no	no	run	run	run	run
1.2	10% (32 μs)	no	no	run	run	run	run
1.3	10% (32 μs)	no	no	run	run	run	run
2.1	50% (160 μs)	no	no	run	run	run	run
2.2	50% (160 μs)	no	no	run	run	run	run
2.3	50% (160 μs)	no	no	run	run	run	run
3.1	90% (288 μs)	no	no	run	run	run	run
3.2	90% (288 μs)	no	no	run	run	run	run
3.3	90% (288 μs)	no	no	run	run	run	run
3.1	130% (416 μs)	no	no	run	run	run	run
3.2	130% (416 μs)	no	no	run	run	run	run
3.3	130% (416 μs)	no	no	run	run	run	run

Table 4: ET execution time delay

Experiment remarks

As the scheduler has a higher priority than the priority of jobs, it preempts the execution of the job_2 after the execution of the first iteration. The scheduler correctly suspends job_2

Let us analyze the first experiment run (Table 4, Exp. 1.1). Job_2 is scheduled in the next core cycle round, and it resumes the execution from the point in time where it was suspended in the previous slot. This means that the job_2 resumes its execution from the last interaction (the last 32 μs of the previous iteration that are caused by the SWIFI) and suspend itself after 32 μs (note that jobs suspend itself after each execution of the periodic part). In this case the 2^{nd} execution of the periodic part of job_2 starts in the 3^{rd} TDMA round, the 3^{rd} execution of the periodic part starts in the 5^{th} TDMA, and so on.

These experiments have shown that a faulty job does not affected the CPU resources allocated to other jobs, and does not affect the correct execution of the XCU_0.

From this analysis and from experiments results we conclude that the EEE fulfills its requirements. An ET job whose execution time is longer than the configured time will not affect the operation of other jobs.

5.3 Spatial partitioning

"A job that suffers from a software fault shall not be able to access the memory areas allocated to other jobs [8]."

5.3.1 Case I: Memory access outside of own address space - code segment

The experiments are executed in XCU_0 which contains three TT jobs (job_1, job_2 and job_3). The cluster is configured such that 320 μs are allocated for execution of each job. The goal was to modify the job_2 by the software fault injection framework such that it tries to write into the memory address of job_1. The fault injection framework shall find out the absolute code space memory addresses of the jobs. A set of experiments will be executed where job_2 systematically tries to access memory areas starting from address *0x0* to the end of the code space.

Expected result

According to the DECOS validation claims, the EEE shall prevent a faulty job (that suffers from a software fault) to access the memory of other jobs. The operation of other jobs and the operation of SCU/XCU shall not be affected.

Experiment result

The operation of other jobs and the operation of the XCU_0 is not affected. The injecting job - the job_2 - overwrites his own code space and gets stuck. Therefore, the spatial partitioning of the EEE is supported in this case.

run#	segment	access prevented	exp. passed	Status job_1	job_2	job_3	system
1.1	code	yes	yes	run	stuck	run	run
1.2	code	yes	yes	run	stuck	run	run
1.3	code	yes	yes	run	stuck	run	run

Table 5: Memory access - code segment

5.3.2 Case II: Memory access outside of own address space - data segment

The experiments are executed in XCU_0 which contains three TT jobs (job_1, job_2 and job_3). The cluster is configured such that 320 μs are allocated for execution of each job. The goal was to modify the job_2 by the software fault injection framework such that it tries to write into the memory address of job_1. The fault injection framework shall find out the absolute data space memory addresses of the jobs. A set of experiments will be executed where job_2 systematically tries to access memory areas starting from address *0x0* to the address *0x100000*.

Expected result

According to the DECOS validation claims, the EEE shall prevent a faulty job (that suffers from a software fault) to access the memory of other jobs. The operation of other jobs and the operation of SCU/XCU shall not be affected.

Experiment result

The operation of other jobs and the operation of the XCU_0 is not affected. The injecting job job_2 overwrites his own data space and continues to run. Therefore, the spatial partitioning of the EEE is supported in this case.

run#	segment	access prevented	exp. passed	Status job_1	job_2	job_3	system
1.1	data	n/a	yes	run	run	run	run
1.2	data	n/a	yes	run	run	run	run
1.3	data	n/a	yes	run	run	run	run

Table 6: Memory access - data segment

5.3.3 Case III: Communication port access violation

"Jobs in the DECOS architecture communicate among each other using *input* and *output ports*. Jobs of time-triggered (TT) DASes access the ports in the predefined points in time. The EEE

shall guarantee that TT jobs do not access their ports outside their partitioned time window [8]."

This experiment can be omitted if it can be proven that the EEE in the DECOS implementation of the Technical University of Vienna successfully prevents the execution of jobs outside their defined partition window. It is evident that if jobs can not run outside their time window, they can not violate the timing constraints on accessing their defined communication ports. As the current Linux-RTAI-LXRT EEE implementation suspends the execution of jobs not finishing in their partition window time in all DASes, either on SCU or XCU node computers, no fault can occur which makes a job of an DAS accesses the I/O port outside their time slot.

5.3.4 Case IV: Memory allocation in user space (multiple)

The experiments are executed in SCU_0 which contains three TT jobs (job_1, job_2 and job_3). Job_2 is modified by the software fault injection (SWIFI) framework such that it tries to allocate memory in the user space by using the call to the RTAI function *rtai_malloc(keyname, size)* with different keyname in the parameters. This ensures that consecutive calls allocate distinct portions of memory.

Expected result

The expected correct result of this test case would be that the function *rtai_malloc(keyname, size)* allocates the specified chunks of memory as long as free chunks are available. The operation of other jobs and the operation of the SCU shall not be affected.

Experiment result

The experiment results are presented in Table 7. Please note that each experiment is executed at least three times.

run#	keyname	size	succ. allocs	Status job_1	job_2	job_3	system
1.1	varying	4 kByte	70	run	run	run	run
1.2	varying	4 kByte	70	run	run	run	run
1.3	varying	4 kByte	70	run	run	run	run
2.1	varying	128 kByte	70	run	run	run	run
2.2	varying	128 kByte	70	run	run	run	run
2.2	varying	128 kByte	70	run	run	run	run

3.1	varying	256 kByte	70	run	run	run	run
3.2	varying	256 kByte	70	run	run	run	run
3.3	varying	256 kByte	70	run	run	run	run
4.1	varying	550 kByte	70	run	run	run	run
4.2	varying	550 kByte	70	run	run	run	run
4.3	varying	550 kByte	70	run	run	run	run
5.1	varying	560 kByte	70	run	run	run	run
5.2	varying	560 kByte	70	run	run	run	run
5.3	varying	560 kByte	70	run	run	run	run
6.1	varying	600 kByte	69	released	released	released	reboot
6.2	varying	600 kByte	69	released	released	released	stuck
6.3	varying	600 kByte	69	released	released	released	reboot
7.1	varying	610 kByte	68	released	released	released	reboot
7.2	varying	610 kByte	68	released	released	released	reboot
7.3	varying	610 kByte	68	released	released	released	reboot
8.1	varying	900 kByte	46	released	released	released	reboot
8.2	varying	900 kByte	46	released	released	released	reboot
8.3	varying	900 kByte	46	released	released	released	reboot
9.1	varying	1 MByte	40	released	released	released	reboot
9.2	varying	1 MByte	40	released	released	released	reboot
9.3	varying	1 MByte	40	released	released	released	reboot

Table 7: Multiple allocations in user space by *rtai_malloc()*, different keynames

5.3.5 Case V: Memory allocation in user space (identical)

The experiments are executed in SCU_0 which contains three TT jobs (job_1, job_2 and job_3). Job_2 is modified by the software fault injection (SWIFI) framework such that it tries to allocate memory in the user space by using the call to the RTAI function *rtai_malloc(keyname, size)* with different keyname in the parameters. This ensures that consecutive calls allocate distinct portions of memory.

Expected result

The expected correct result of this test case would be that the function *rtai_malloc(name, size)* allocates the specified chunk of memory at the first

run and gets a reference to the address at all subsequent runs. The operation of other jobs and the operation of the *SCU* shall not be affected.

Experiment result

The experiment results are presented in Table 8. Please note that each experiment is executed at least three times.

run#	keyname	size [kBytes]	succ. allocs	Status job_1	job_2	job_3	system
1.1	equal	128	24567	run	run	run	run
1.2	equal	128	24567	run	run	run	run
1.3	equal	128	24567	run	run	run	run
2.1	equal	256	12283	run	run	run	run
2.2	equal	256	12283	run	run	run	run
2.3	equal	256	12283	run	run	run	run
3.1	equal	512	6140	run	run	run	run
3.2	equal	512	6140	run	run	run	run
3.3	equal	512	6140	run	run	run	run
4.1	equal	1024	3069	run	run	run	run
4.2	equal	1024	3069	run	run	run	run
4.3	equal	1024	3069	run	run	run	run

Table 8: Multiple allocations in user space by *rtai_malloc()*, constant keyname

Experiment remarks

The job allocates the chunk of memory one time but gets no reference after a certain amount of calls (see Table 8 in column "successful allocs").

5.3.6 Case VI: Memory allocation in kernel space (multiple)

The experiments are executed in SCU_0 which contains three TT jobs (job_1, job_2 and job_3). Job_2 is modified by the software fault injection framework such that it tries to allocate memory in the kernel space by using the call to the RTAI function *rtai_kmalloc(name, size)* with different names for the parameters key. This ensures that consecutive calls allocate distinct portions of memory.

Expected result

The expected correct result of this test case would be that the function *rtai_kmalloc(name, size)* allocates the specified chunks of memory as long as free chunks of memory are available. The operation of other jobs and the operation of the *SCU* should not be affected.

Experiment result

Job_2 in the SCU_0 makes a call to the RTAI function *rtai_kmalloc(name, size)* with different keyname in the parameter. The experiment gets repeated for increasing values for the parameter size to observe different behavior. The result are presented in Table 9.

run	key name	size [kByte]	succc. allocs	Status job_1	job_2	job_3	system
1.1	varying	4	70	run	run	run	run
1.2	varying	4	70	run	run	run	run
1.3	varying	4	70	run	run	run	run
2.1	varying	256	70	run	run	run	run
2.2	varying	256	70	run	run	run	run
2.3	varying	256	70	run	run	run	run
2.1	varying	512	70	run	run	run	run
2.2	varying	512	70	run	run	run	run
2.3	varying	512	70	run	run	run	run
3.1	varying	1024	40	released	released	released	reboot
3.2	varying	1024	40	unknown	released	unknown	stuck
3.3	varying	1024	40	released	released	released	reboot

Table 9: Multiple allocations in kernel space by *rtai_kmalloc()*, different keynames

5.3.7 Case VII: Memory allocation in kernel space (identical)

The experiments are executed in SCU_0 which contains three TT jobs (job_1, job_2 and job_3). Job_2 is modified by the software fault injection framework such that it tries to allocate memory in the kernel space by using the call to the RTAI function *rtai_kmalloc(name, size)* with a constant names for the parameter key. This ensures that consecutive calls allocate the same portion

of memory.

Expected result

The expected correct result of this test case would be that the function *rtai_kmalloc(name, size)* allocates the specified chunks of memory on the first call and returns a reference to the former allocated memory for consecutive calls. The operation of other jobs and the operation of the *SCU* should not be affected.

Experiment result

Job_2 in the SCU_0 makes a call to the RTAI function *rtai_kmalloc(name, size)* with a constant keyname for the parameter *key*. The experiment gets repeated for increasing values for the parameter size to observe different behavior. The result are presented in Table 9.

run#	keyname	size [kBytes]	succ. allocs	Status job_1	job_2	job_3	system
1.1	equal	128	24567	run	run	run	run
1.2	equal	128	24567	run	run	run	run
1.3	equal	128	24567	run	run	run	run
2.1	equal	256	12283	run	run	run	run
2.2	equal	256	12283	run	run	run	run
2.3	equal	256	12283	run	run	run	run
3.1	equal	512	6140	run	run	run	run
3.2	equal	512	6140	run	run	run	run
3.3	equal	512	6140	run	run	run	run
4.1	equal	1024	3069	run	run	run	run
4.2	equal	1024	3069	run	run	run	run
4.3	equal	1024	3069	run	run	run	run

Table 10: Multiple allocations in kernel space by *rtai_kmalloc()*, constant keyname

5.3.8 Case VIII: Memory allocation in the heap

In this experiment, job_2 tries to allocate a series of 1kB, 128kB, 256kB, 512kB, 1MB of memory in the heap space using the native C function *mal-*

loc() once per task activation.

Expected result

The expected correct result of this test case would be that the function *malloc()* allocates the specified chunks of memory as long as free chunks are available, and the operation of other jobs and the operation of the SCU/XCU shall not be affected.

Experiment result

Job_2 continues to execute until the whole free memory of the computer is allocated, and afterwards is terminated by the operating system. In this case the operation of other jobs is not affected, and the memory that was allocated by the faulty job is released. This experiment run is repeated where the faulty job tries to allocate the memory in chunks of 1kB, 128kB, 256kB, 512kB, 1MB. The results are presented in Table 11.

run#	size [kByte]	succc. allocs	Status job_1	job_2	job_3	system
1.1	1	41704	running	released	running	reboot
1.2	1	41704	running	released	running	running
1.3	1	41704	running	released	running	running
2.1	128	318	running	released	running	running
2.2	128	318	running	released	running	reboot
2.3	128	318	running	released	running	running
3.1	256	161	running	released	running	reboot
3.2	256	161	running	released	running	running
3.3	256	161	running	released	running	reboot
4.1	512	81	running	released	running	stuck
4.2	512	81	running	released	running	running
4.3	512	81	running	released	running	reboot
5.1	1024	40	running	released	running	stuck
5.2	1024	40	running	released	running	reboot
5.3	1024	40	running	released	running	reboot

Table 11: Memory allocation in the heap by *malloc()*

The outcome of the experiments was in most of the cases the termination of job_2. However the behavior of the node computer was not deterministic, as in some cases the node computer was still executing the jobs job_1 and job_3, and in some experiment runs it either stuck or has caused that the whole node rebooted. Therefore, in this case the EEE does not fulfill the DECOS requirements, as a faulty job (job_2) can affect the operation of the SCU/XCU.

5.3.9 Case IX: Memory allocation in the heap

This experiment should test the allocation in the heap memory space by executing the function *rt_malloc()* a defined number of times. The test could not be performed, as the required RTAI module `rt_mem_mgr` supporting this type of memory allocation is not included in the DECOS implementation of the Technical University of Vienna.

5.3.10 Case X: Stack overflow by recursive functions

The experiments are conducted with the same setup as in Case I (see Section 5.2.1). Job_2 is modified by the software fault injection framework such that it calls a recursive function. In this case the length of the stack grows with every recursive call. The recursive function is listed below:

```
long recursivefunction(long iteration){
   volatile long garbage[1024];
   garbage[1024-1]=rand();
   rt_printk("i: %d\n",iteration++);
   garbage[1024-2] = recursivefunction (iteration);
   return 0;
}
```

Expected result

According to the DECOS validation claims, a job which suffers from software faults shall not be able to affect the operation of other jobs or the operation of the SCU/XCU.

Experiment result

The experiment results are presented Table 12. Please note that each experiment is executed at least three times. The experiments are executed with the recursive function that declares different array sizes.

run#	size [kByte]	succc. recursions	Status job_1	job_2	job_3	system
1.1	4	503	run	released	run	run
1.2	4	503	run	released	run	run
1.3	4	503	run	released	run	run
2.1	32	62	run	released	run	run
2.2	32	62	run	released	run	run
2.3	32	62	run	released	run	run
3.1	64	30	run	released	run	run
3.2	64	30	run	released	run	run
3.3	64	30	run	released	run	run
4.1	4096	0	run	released	run	run
4.2	4096	0	run	released	run	run
4.3	4096	0	run	released	run	run

Table 12: Stack-overflow by recursive function call

The operation of other jobs was not affected and operation of the application computer subsystem was not affected as well. In this case the EEE fulfills the DECOS validation requirements.

5.3.11 Case XI: Manipulation of the scheduler semaphore

The synchronization of the scheduler is performed by the semaphore `LOCK`. On each start of a core cycle, the network device driver performs a *sem_v()* operation to unblock the scheduler task which in term triggers the application jobs to be run in the configured time slot.

The experiments are executed in XCU_0 which contains three TT jobs. Job_2 tries to access the schedulers semaphore `LOCK` and perform consecutive *sem_v()* operations.

Expected result

The EEE shall provide a mechanism robust enough to detect such access and perform countermeasures to guarantee the timely correct behavior.

Experiment result

It is possible to access the semaphore from within an application job. Con-

secutive *sem_v()* operations from an application job cause the scheduler to never block. This causes multiple executions of the jobs within the configured partition window time. The possibility to access the scheduler's semaphore can cause to break the temporal behavior of a DECOS unit.

6 Experiment analysis

In this section the analysis of the experiments executed in Section 5 in the temporal and spatial domain are presented. Special system calls executed on the experiment runs are described and how they are executed in the Linux-RTAI-LXRT environment.

6.1 Temporal encapsulation

This section covers the experiments executed in Section 5.2 concerning deadline violations of TT jobs and delayed ET jobs.

6.1.1 Deadline Violation of TT jobs

Experiment analysis

The TT task job_2 is scheduled at the beginning of the first TDMA round. At the end of the partition window time, the TT task gets interrupted by the scheduler. Let us analyze the first experiment run (Table 2, Exp. 1.1). Job_2 is scheduled in the next TDMA round, and it resumes the execution from the point in time where it was suspended in the previous slot. This means that the job_2 resumes its execution from the last interaction (the last 32 μs of the delay of the previous iteration that are caused by the SWIFI) and suspend itself after 32 μs (note that jobs suspend itself after each execution of the periodic part). In this case the 2^{nd} execution of the periodic part of job_2 starts in the 3^{rd} TDMA round, the 3^{rd} execution of the periodic part starts in the 5^{th} TDMA, and so on. Figure 19 and Figure 20 show the job timings if a jobs execution time is delayed to $> 100\%$ and $> 300\%$ of its allocated time slot. The activity of job_2 is depicted as the 3^{rd} signal from the top.

These experiments have shown that a faulty job does not affected the CPU resources allocated to other jobs, and does not affect the correct execution of the SCU. However, the faulty job was not terminated as it is required from the EEE. Job_1 and job_3 are executed in their allocated time slot, and its operation is not affected by the execution of the faulty job job_2. From this analysis and from experiments results we conclude that the EEE partially fulfills its requirements. A fault job TT will not affect the operation of other jobs, but the faulty TT job gets not terminated.

As the scheduler has a higher priority than the priority of application jobs, it preempts the execution of the job_2 after the execution of the first iteration. The scheduler also suspends job_2 during the execution of the infinite loop as

shown in Section 5.2.2. However the scheduler cannot terminate the execution of this job, as the application jobs in the current DECOS framework are implemented as LXRT task, and there is no reliable possibility to terminate an LXRT task.

The current implementation of the scheduler makes no distinction whether a task is a ET or TT task. The scheduler could be implemented in a way that a TT task executing longer than the configured partition time window would be terminated, but the problem is that all application jobs in the DECOS implementation of the Technical University of Vienna are LXRT tasks [15]. LXRT tasks can not be terminated in a reliable way, if the tasks do not respond. The problem is that real-time tasks in userspace (LXRT tasks) have a "buddy"-task in the kernel space, which in term executes RTAI-functions if called. The termination of an LXRT implemented application job requires the termination of the kernel module ("buddy"-task) *and* the LXRT task. Should the LXRT task not respond to a request for termination, only the kernel task can be killed, and a *zombie* LXRT task is left in the system. This prevents a re-launch of the application job after termination as the RTAI object, which is created for each task, each shared memory, and each FIFO is already instantiated. In the case of an unresponsive LXRT task, only a reset of the node computer can lead to an correct system state.

As the interruption of the job that violates its deadlines is not observed, no information are sent to the diagnosis high-level service (symptom generation). However, a proper implemented diagnosis service will detect that the output port was not updated, and such a situation can be correctly detected by the diagnosis as a software fault.

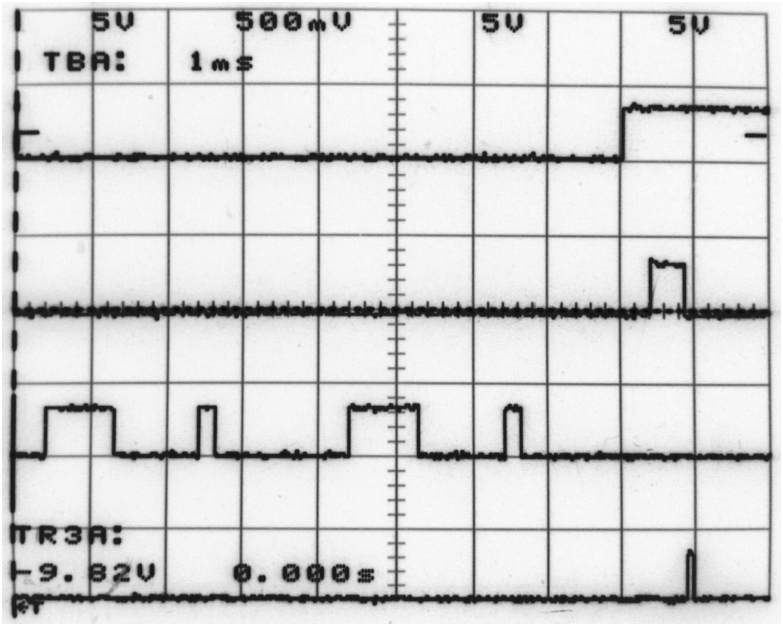

Figure 19: Job execution time > 100% of allocated time slot

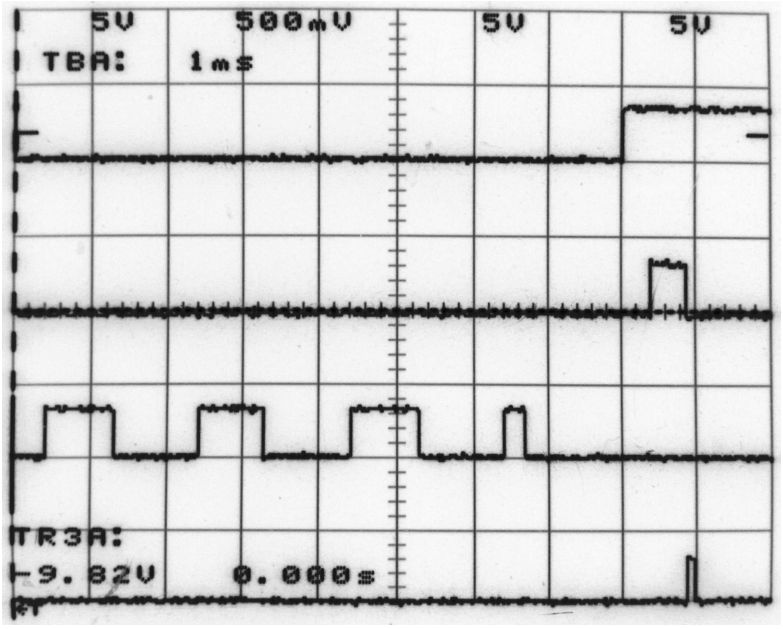

Figure 20: Job execution time > 300% of allocated time slot

6.2 Spatial partitioning

This section covers the experiments executed in Section 5.3 concerning the spatial encapsulation in the DECOS implementation of the Technical University of Vienna.

6.2.1 Memory access outside of own address space violation

The experiment shall perform a write access of a DECOS job into a memory partition of another job.

Experiment analysis

In this case two situations can be observed: the case when the job accesses the data segment, and the case when the job accesses the code segment. The results of experiments for the code segment are shown in Table 5. The corresponding result set for the data segment are presented in Table 6.
An attempt of job_2 to access the memory addresses to other jobs will fail, because the concept of virtual memory and the memory protection mechanisms of the Linux OS prevent a job running in the user mode to write in the memory areas of other jobs.

In case that the job accesses its own data segment, the job will not be interrupted. In case the job accesses the own code space, the job will be interrupted by the EEE, however the operation of other jobs is not affected and the operation of the XCU_0 is not affected. Therefore, the spatial partitioning of the EEE is supported in this case.

Remarks on memory manipulation of foreign jobs

The results showed that it is not possible to write to foreign data sections. As the jobs are independent RTAI tasks, and the scheduler can only suspend and resume these, it is generally not possible to overwrite foreign memory regions. The Linux OS has sufficient security checks on memory access functions which makes it impossible. This had already been researched and argumented in [15]. The SWIFI framework used kernel hacking and duplicating of kernel source code into the DECOS framework to make it possible to inject a byte into code and data segments. The "attacker" must be able to load a kernel module into the OS and have detailed knowledge of the Linux version used by the node.

As a short summary, one can say it is theoretically possible to corrupt another job's memory, but therefore the developer has to have the facility to

load a kernel module and design his application job for that specific reason. For further information about conditions a job has to fulfill, the reader is advised to read [15, 16, 17]

6.3 System calls

This section covers the execution of RTAI and Linux system calls from within a DECOS job. The effects on invocation of such system calls are presented.

6.3.1 Memory allocation in user space

In this experiment class, consecutive calls of the function *rtai_malloc* are performed on each activation of a DECOS job job_2 in the unit SCU_0 to allocate memory in the user space (see Section 5.3.4 and 5.3.5).

Experiment remarks
Two observations could be done in this result:

- There is an upper limit of 70 iterations of successful allocation runs before request for memory chunks get denied by the system
- The upper memory limit for allocations is 41 MB before the allocating task gets stuck. This is due the limited amount of RAM available in the DECOS units SCU and XCU

The first observation could be pinned down for the fact that in RTAI's LXRT mode each object, namely

- TSK - tasks
- SEM - named semaphores
- RWL - read write locks
- SPL - spin lock
- MBX - mailbox
- PRX - proxy

is stored in global RTAI object list. This list has a limited amount of entries which is defined by the macro `MAX_SLOTS` which in LXRT is a define to `CONFIG_RTAI_SCHED_LXRT_NUMSLOTS` and is set to 100 in the current DECOS

implementation. This value can only be changed at compile-time, not dynamically on run-time. As soon as the table is full, all requests for additional objects are blocked by the RTAI framework.

At experiment runs 1.1 to 5.3 (see Table 7) job_2 tries to allocate shared memory in the user space using *rtai_malloc()* once in a TDMA round. Shared memory can be allocated in junks of 4 kB, i.e., this is the minimum size for one allocation. After the execution of 70 rounds job_2 cannot allocate memory, as for each allocation a shared memory object is generated, as the limit of maximum objects that can be allocated is reached. After the maximum number of generated object is reached (i.e., after 70 allocations) the job fails to allocate more memory. However, the function that performs allocation (*rtai_malloc()*) does not give an error return value when it fails to allocate the shared memory in the user space. Job_2 continues to operate, and neither its operation nor the operation of other jobs is affected even if job_2 tries to further allocate shared memory blocks.

At experiment runs 6.1 to 9.3 (see Table 7) job_2 tries to allocate shared memory in the user space using *rtai_malloc* once in a TDMA round. After the execution of 69 TDMA rounds, for experiment run 6.1 and 40 TDMA rounds for experiment run 9.3, job_2 has allocated the maximum free memory. The operating system terminates the faulty job, however the operation of other jobs is affected as well as the SCU performs a reboot, and in one case gets stuck. Therefore in this case the EEE does not fulfill the requirement that it should protect other jobs by being affected by a job that suffers from a software fault.

Similar experiments are repeated where the keyname (ID) of the requested memory allocation remain the same. In this case, the attempt to allocate memory with the same keyname, will result in the allocation of the same memory. In case that the memory of different size is allocated with the same keyname, the size of the allocated memory will results in the size of the last allocation. In this case the operation of the job performing memory allocation is not affected, as it is not affected the operation of other jobs and of the SCU/XCU.

6.3.2 Memory allocation in kernel space

In this experiment class, consecutive calls of the function *rtai_kmalloc()* are performed on each activation of a DECOS job job_2 in the unit SCU_0 to allocate memory in the kernel space (see Section 5.3.6 and 5.3.7).

Experiment analysis

The observation from this experiment is the same as from the experiment allocating memory from the user space by using the *rtai_malloc()* operation, so one could assume the underlying system calls could be exactly the same. Looking into `rtai_shm.h` it gets clear that the call to *rtai_kmalloc()* in LXRT-mode is actually just a macro that allocates a chunk of the shared memory like the call to *rtai_malloc()*. The shared memory pool for RTAI gets pre-reserved on the initialization of the RTAI subsystem, i.e. loading of the RTAI kernel modules. Necessary increase of this memory pool gets done dynamically at runtime when the control (the CPU) is handed over from the RTAI to the Linux operating system, as the allocation is performed by Linux system calls and these are blocking functions. For the dynamic increase of the memory available, the RTAI module `rt_mem_mgr` is required to be loaded. In the current DECOS implementation this module is not available, so dynamic increase of the memory pool at runtime is not possible.

The following is the define macro in the file `rtai_shm.h` for the function *rtai_kmalloc()*.

```
#define rtai_kmalloc(name, size) \
rt_shm_alloc(name, size, USE_VMALLOC)  // legacy

#define rt_shm_alloc(name, size, suprt)  \
_rt_shm_alloc(0, name, size, suprt, 0)



#define rtai_malloc(name, size)  \
_rt_shm_alloc(0, name, size, USE_VMALLOC, 0)  // legacy
```

We can see, *rtai_kmalloc* is actually just a call to the function *rt_shm_alloc()*. Comparing to the way on how *rtai_malloc()* is called, we can see there is no difference in the invocation scheme. The two functions can therefore be seen as the same in the RTAI framework. The calling mechanism is described in Figure 21 and shown in the listing below. Each call checks, whether the RTAI-name given to the function already exists in the global RTAI object list. If the RTAI-name is found, a reference to that memory location is returned. If the name can not be found, a new memory section is allocated and returned to the calling function. For the case of an LXRT task, the memory is allocated in the user space, whereas a call from a kernel module allocates the memory in the kernel space.

```
static inline void *_rt_shm_alloc(unsigned long name, int size, int suprt)
{
```

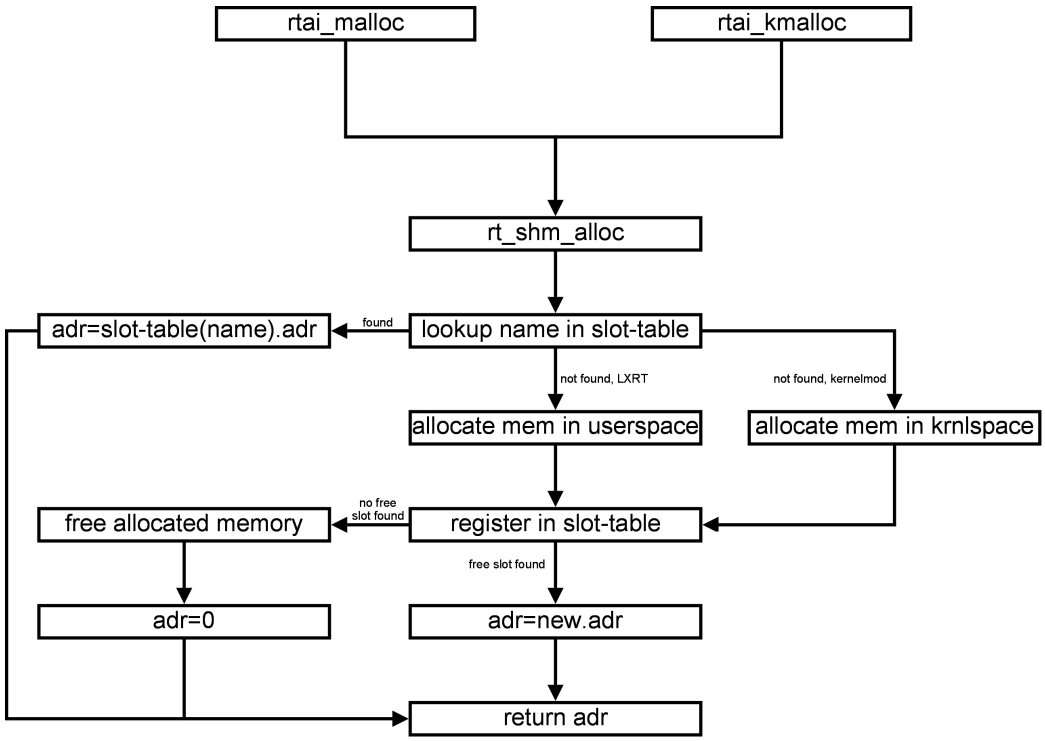

Figure 21: RTAI memory allocation

```
  void *adr;
  // suprt = USE_GFP_ATOMIC; // to force some testing
  if (!(adr = rt_get_adr_cnt(name)) && size > 0 && suprt >= 0 \
           && RT_SHM_OP_PERM()) {
    size = ((size - 1) & PAGE_MASK) + PAGE_SIZE;
    if ((adr = suprt ? rkmalloc(&size, SUPRT[suprt]) : rvmalloc(size))) {
      if (!rt_register(name, adr, suprt ? -size : size, 0)) {
        if (suprt) {
          rkfree(adr, size);
        } else {
          rvfree(adr, size);
        }
        return 0;
      }
      memset(ALIGN2PAGE(adr), 0, size);
    }
  }
  return ALIGN2PAGE(adr);
}
```

Allocations limited by the global RTAI object list

The number of successful invocations of RTAI memory allocations is limited by the amount of available entries in the global object list. The number of 70 available entries is due to the fact that tasks, FIFO's and shared memory regions require an entry in the list, and the DECOS unit is configured for a total number of 100 objects. 30 objects are already used after startup. The total list of RTAI objects instantiated in the memory space can be derived from the command "`tail -n 50 /proc/rtai/RTAI names`" at the console shell:

```
RTAI LXRT Information.

    MAX_SLOTS = 100

                                        Linux_Owner         Parent PID
Slot Name   ID         Type   RT_Handle    Pointer   Tsk_PID   MEM_Sz   USG Cnt
-------------------------------------------------------------------------------
   1 RTGLBH 0x9ac6d9e5 SHMEM  0x00000000 0x00000000        0     131072       1
   2 XCHNGE 0xb8b3f8f7 SHMEM  0xc4871000 0x00000000        0       4096       4
   3 CONF   0x000c5c8c SHMEM  0xc4873000 0x00000000        0       4096       2
   4 TKDATA 0xa4479cc1 SHMEM  0xc4875000 0x00000000        0       4096       3
   5 LOCK   0x00148053 SEM    0xc48b2880 0x00000000        0          0       1
   6 THLSVC 0xa3e55f18 MBX    0xc3760040 0x00000000        0          0       1
   7 TJOB1  0x0435b309 MBX    0xc37600c0 0x00000000        0          0       1
   8 TJOB2  0x0435b30a MBX    0xc3760140 0x00000000        0          0       1
   9 TJOB3  0x0435b30b MBX    0xc37601c0 0x00000000        0          0       1
  10 FIMBXF 0x58bf371e MBX    0xc3760240 0x00000000        0          0       1
```

```
  11 FIMBXS 0x58bf372b MBX    0xc37602c0 0x00000000        0          0       1
  12 FIMBXC 0x58bf371b MBX    0xc3760340 0x00000000        0          0       1
  13 SHMBUF 0x9e852b63 SHMEM  0xc4893000 0x00000000        0      12288       2
  14 HLSVC  0x029002c6 TASK   0xc3763040 0x00000000        0        558       1
  15 0BH    0x000007d7 SHMEM  0xc4857000 0x00000000        0       4096       1
  16 0EA    0x00000845 SHMEM  0xc4880000 0x00000000        0       4096       2
  17 0EB    0x00000846 SHMEM  0xc4897000 0x00000000        0       4096       2
  18 0EC    0x00000847 SHMEM  0xc4899000 0x00000000        0       4096       2
  19 JOB1   0x0012b0ab TASK   0xc3763840 0x00000000        0        560       1
  20 FITEST 0x58c59e3d SHMEM  0xc489b000 0x00000000        0       4096       2
  21 FITIME 0x58c5b508 SHMEM  0xc489d000 0x00000000        0       4096       2
  22 SM_LOG 0x9f42924b SHMEM  0xc489f000 0x00000000        0       4096       3
  23 T4_LNK 0xa227e09d SHMEM  0xc48b5000 0x00000000        0       4096       2
  24 JOB2   0x0012b0ac TASK   0xc3764040 0x00000000        0        562       1
  25 T_IVAR 0xa6816b55 SHMEM  0xc48b7000 0x00000000        0       4096       2
  26 T_OVAR 0xa686d99f SHMEM  0xc48b9000 0x00000000        0       4096       2
  27 T_EXCH 0xa67dd89f SHMEM  0xc48bb000 0x00000000        0       4096       2
  28 JOB3   0x0012b0ad TASK   0xc3764840 0x00000000        0        564       1
  29 FINJH  0x0246925b TASK   0xc3765040 0x00000000        0        572       1
  30 FINJHT 0x58c04bfb TASK   0xc3765840 0x00000000        0        577       1
```

The following is the console output on startup of the node computer, where we can see the name of the RTAI objects of the LXRT tasks and the job's addresses in memory:

```
This is node (00:00:24:C3:C1:94) ...
Component: 0
SCU (subsystem 0)

HLSVC: Task init, address = 0xc3763040.
VN at Component 0 and Subsystem 0
 * comfort CAN
 * AVDN_NSC
 * PVDN_NSC
 * PVDN_SC
 * lights DAS
 * navigation DAS
 * by wire DAS
 * Lights PCAN
 * Comfort LIN
 * BB4 Application
 * VAL Application
GW at Component 0 and Subsystem 0
 * GW "VGW:CAN0_to_TT" located at subsystem 0, component 0
Slot 0, task: c3763040, taskname: HLSVC
JOB1: Task init, address = 0xc3763840.
Slot 0, task: c3763040, taskname: HLSVC
```

```
Slot 1, task: c3763840, taskname: JOB1
JOB2: Task init, address = 0xc3764040.
Slot 0, task: c3763040, taskname: HLSVC
Slot 1, task: c3763840, taskname: JOB1
Slot 2, task: c3764040, taskname: JOB2
JOB3: Task init, address = 0xc3764840.
Slot 0, task: c3763040, taskname: HLSVC
Slot 1, task: c3763840, taskname: JOB1
Slot 2, task: c3764040, taskname: JOB2
Slot 3, task: c3764840, taskname: JOB3
SCHED: All tasks registered
```

6.3.3 Manipulation of the scheduler semaphore

This experiment aims at the RTAI semaphore access from within jobs. Allowing unprotected access to RTAI objects - shared memory or semaphores - from application jobs may be very dangerous. This DECOS implementation uses RTAI shared memory for the most critical sections, namely the

- configuration storage (`"CONF"`),
- scheduling (`"LOCK"`) and
- cluster communication (`"XCHNGE"`)

Manipulating those shared memory regions from within an application job can lead to unpredictable results.

In the current DECOS implementation of the Technical University of Vienna, the semaphore "LOCK" is used to synchronize the scheduler task with the RTnet network device driver.

The cluster starts up by initializing the scheduler task, which does the initialization with RTAI semaphore `"LOCK"` creation and then schedules itself for the normal operation to schedule the jobs corresponding to their position in the TDMA cycle. The actual schedule is an infinite loop which resumes the tasks and waits for the semaphore as the last instruction in the loop. The next iteration of the loop gets triggered on receiving an incoming message at the network interface (see Figure 22). As soon as a message arrives at the network interface, the interrupt handler of the device driver signals the semaphore `"LOCK"` so that the scheduler is informed that the execution instant has occurred.

If a malicious application job signals this semaphore, the scheduler would

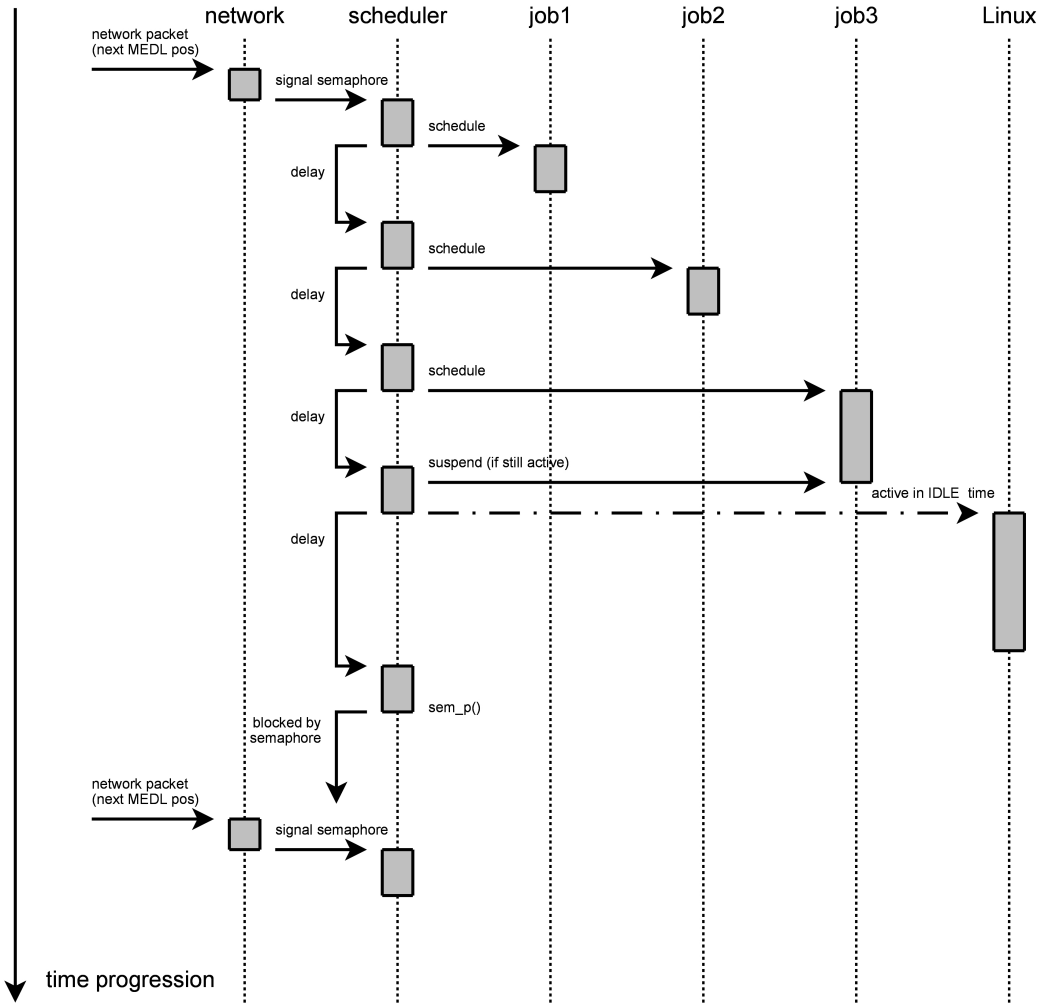

Figure 22: DECOS scheduling

run instantly which could provoke an unpredictable state in the unit.

The same error state could occur if any task of a node deletes the semaphore: The RTAI documentation describes on deletion of semaphores by calling *rt_sem_delete(SEM *sem)*:

> "Any tasks blocked on this semaphore is returned in error and allowed to run when the semaphore is destroyed [18]."

This would mean, the scheduler would continuously execute the infinite scheduling loop with the high priority. A simplified version of the scheduler can be seen in the following source code fragment:

```
static int __init initialize(void) {
    do_initialization_code();
    /* initialize semaphore */
    rt_sem_init(&sem, 0);
    rt_register(nam2num("LOCK"), &sem, IS_SEM, 0);
    do_further_initialization_code();
    rt_set_oneshot_mode(); // set rt_timer to one_shot_mode
    start_rt_timer(0);     // start rt_timer
    rt_task_resume(&sched_task);
    return 0;
}

static void scheduler(int dummy) {
    do_initialization_code();
    while(1) {
        execute_tasks_of_current_slot_and_goto_sleep();
        /* wait for new slot --> semaphore is signaled
           by device driver */
        /* The following signal is generated in
           rt_net-source/drivers/natsemi/rt_natsemi.c */
        rt_sem_wait(&sem);
    }
}
```

The device driver source code signaling the semaphore can be seen below:

```
static void intr_handler(unsigned int irq, void *dev_instance){
    do_interrupt_code();
    if (intr_status &
            (IntrRxDone | IntrRxIntr | RxStatusFIFOOver |
        IntrRxErr  | IntrRxOverrun)) {
    netdev_rx(dev, &time_stamp);
    }
    do_further_interrupt_code();
```

```
}

static void netdev_rx(struct rtnet_device *dev, rtos_time_t *time_stamp){
    do_interrupt_code();
    if ((sem = rt_get_adr(nam2num("LOCK")))) {
        rt_sem_signal(sem);
    }
```

Experiment result

The following figures show the measurements with an oscilloscope. The actual state of the different tasks *scheduler* and job_2, whether running or blocked have been put on output pins of the Soekris node. The oscilloscope shows the actual timing of the unit after performing the fault injection with the tampered semaphore. The upmost signal shows the alternating TDMA round. The middle signal shows the activation of the scheduler task, and the signal on the bottom of the picture shows the activation times of job_2. It is evident that the job_2 is executed too often, as it normally should by activated only six times per TDMA round. The manipulated semaphore never causes the scheduler to block: immediately after finishing a scheduler loop, the next iteration of the scheduling algorithm starts over.

Triggering of the semaphore from a job

The Figure 23 shows the effect on triggering the semaphore in the unit SCU_0 from the job job_2. The scheduler is activated even if no network packet for synchronization is received.

Deletion of the semaphore from a job

The Figure 24 (and Figure 25 in a more detailed resolution) show the effect on deleting the semaphore in the DECOS unit SCU_0. The scheduler is constantly triggered when the semaphore is deleted.

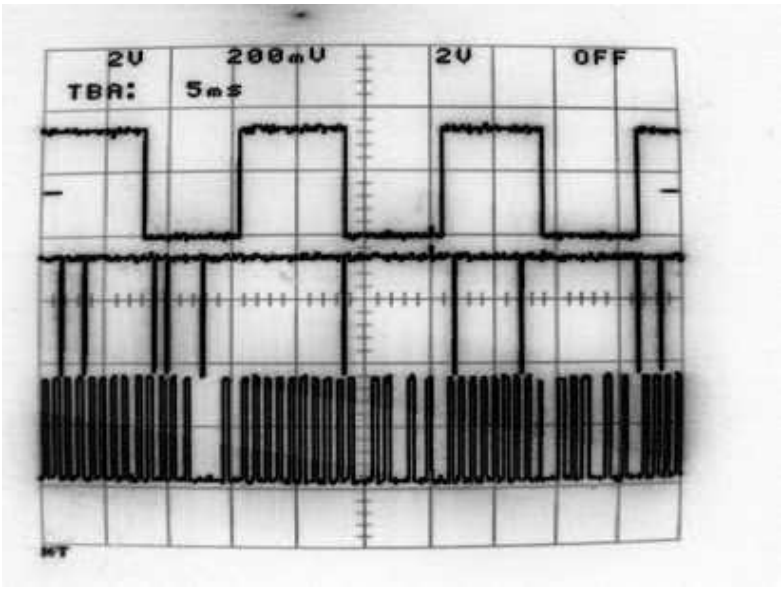

Figure 23: Scheduler operation with manipulated semaphore

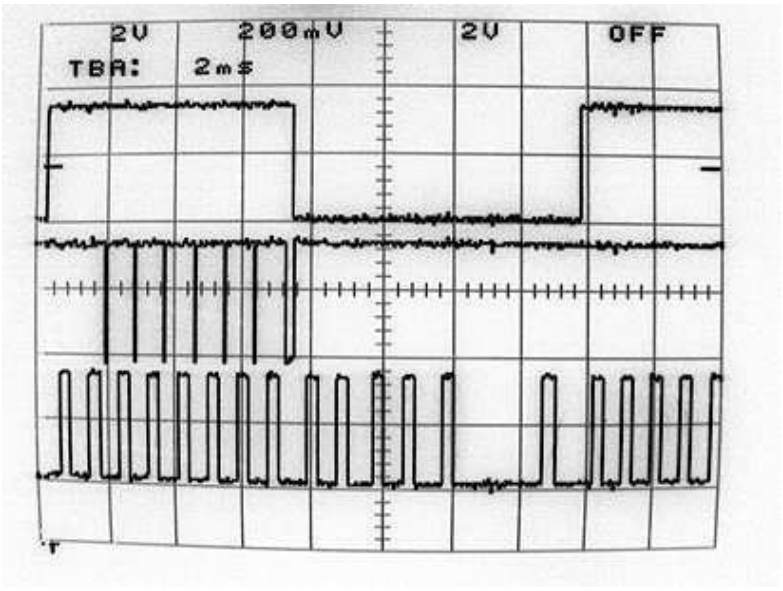

Figure 24: Scheduler operation on deleted semaphore

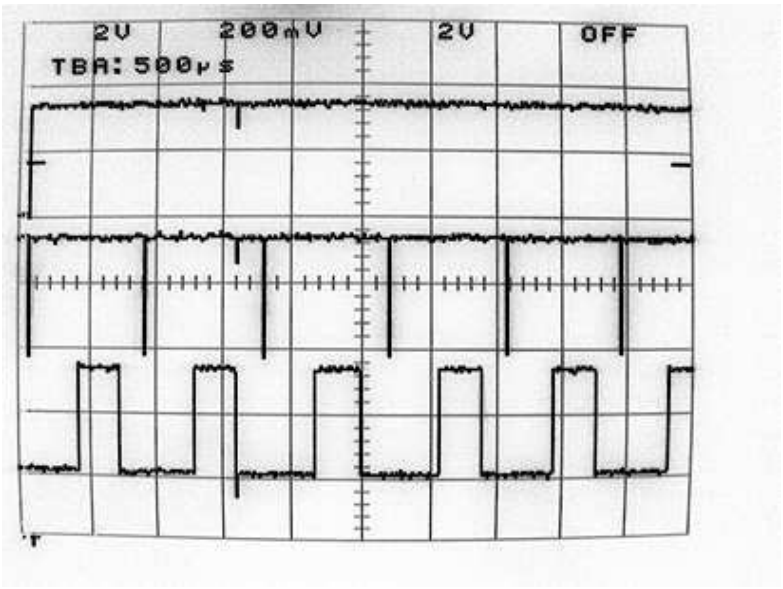

Figure 25: Scheduler operation on deleted semaphore, detailed

7 Conclusion

The DECOS integrated architecture is developed in the course of the European IST-FP6 Integrated Project DECOS. It allows the integration of several embedded application subsystems with different criticality into the same hardware infrastructure. The DECOS Encapsulated Execution Environment (EEE) implemented at the University of Technology Vienna was validated by means of software implemented fault injection (SWIFI). This section presents concluding remarks regarding the implementation of the SWIFI framework for the validation of the DECOS EEE, the test results, and an analysis of the test results.

7.1 Achievements

The work presented in this thesis was performed in the course of the DECOS project. An existing framework for the emulation of hardware faults in the DECOS integrated architecture was extended. A fault injection framework capable to perform the injection of software faults into a DECOS unit was developed. This framework was used to validate the DECOS encapsulated execution environment (EEE) w.r.t. SW faults of the DECOS fault hypothesis. The DECOS EEE is responsible for the partitioning of shared resources (CPU, memory, and I/O) and the encapsulation of multiple DECOS jobs in the temporal and the spatial domain. The extension provides a mechanism to inject software faults by modifying the application jobs. Experiments for the validation of the *temporal* and *spatial* encapsulation were executed.

The Linux-RTAI-LXRT real-time framework is used for temporal and spatial encapsulation in the presented DECOS EEE implementation. RTAI-LXRT can guarantee a correct temporal encapsulation of the DECOS jobs, although the validation of the current scheduling mechanism implementation unveiled two problems:

- The current scheduler implementation is not capable to detect deadline violations of TT jobs. Furthermore, there are no mechanisms to detect and terminate a faulty TT job that violates its deadline. At least, the operation of other jobs executing on the same unit is not affected.
- It is possible to gain unprotected access to a core component of the scheduler, the synchronization semaphore. This mechanism is used to synchronize the application task activations with the DECOS core communication subsystem time slots. A faulty application job can tamper the semaphore and cause the scheduler to execute at instants

outside the specified time slots. The development of a proper protection mechanism for the semaphore is required to maintain the temporal correctness.

Regarding the *spatial* encapsulation, the memory protection mechanisms of the Linux operating system can guarantee a successful spatial encapsulation. A job in a DECOS unit cannot gain access to a partition (assigned memory region) of another job. Problems arose, when system calls of the RTAI-LXRT real-time framework have been invoked from within a DECOS job:

- The dynamic memory allocation functions, available to RTAI-LXRT jobs, cannot be classified as safe. Allocations in kernelspace or userspace can affect the operation of the DECOS unit in a way that the unit fails as a whole. According to the DECOS fault hypothesis on software faults, each application job forms a fault containment region. A unit failing as a whole would therefore not be detected as an application fault by a proper implemented diagnosis service, but as a hardware fault. A limitation of the API set regarding dynamic memory allocation provided to DECOS jobs would be required to overcome this problem.

- Regarding the protection of the input/output ports of DECOS jobs, the same problem like with the synchronization semaphore arose. The CNI in the current implementation is designed as an RTAI shared memory region. A DECOS job getting hold of the respective RTAI object name can gain full access to the communication network interface (CNI) without invoking the functions of the DECOS interface, which shall provide protection mechanisms.

At the current stage of the implementation, the Linux-RTAI-LXRT design does not fulfill all requirements to be deployed as a DECOS EEE. Further improvements with respect to the scheduler mechanism, the protection of the CNI and the limitation of the available API set to static functions have to be implemented to make the current implementation suitable for deployment.

References

[1] H. Kopetz, Design Principles for Distributed Embedded Applications, Kluwer Academic Publishers, 6th printing, 2002.

[2] H. Kopetz, R. Obermaisser, P. Peti and N. Suri, From a federated to an integrated architecture for dependable real-time embedded systems, 2004.

[3] H. Kopetz, A. Ademaj, P. Grillinger, K. Steinhammer, The Time-Triggered Ethernet (TTE) Design, 2005.

[4] R. Obermaisser, P. Peti and H. Kopetz, Virtual Networks in an Integrated Time-Triggered Architecture, 2005.

[5] R. Obermaisser, P. Peti and H. Kopetz, Virtual Communication Links and Gateways - Implementation of Design Tools and Middleware Services, 2005.

[6] W. Herzner, DECOS - PIL Structure Design and API Specification, 2005.

[7] DECOS Team, DECOS - Dependable Embedded Components and Systems, http://www.decos.at/, 2008.

[8] A. Ademaj, DECOS - Component validation, Doc. Id.: 4.2.4-1 Version 1.0, 2007.

[9] Z. Segall, D. Vrsalovic, D. Siewiorek, D. Yaskin, J. Kownacki, J. Barton, R. Dancey, A. Robinson, T. Lin, FIAT-fault injection based automated testing environment, June 1988.

[10] J.H. Barton, E.W. Czeck, Z.Z. Segall, D.P. Siewiorek, Fault injection experiments using FIAT, Computers, IEEE Transactions on Volume 39, Issue 4, April 1990.

[11] S. Han, K.G. Shin, H.A. Rosenberg, DOCTOR: an integrated software fault injection environment for distributed real-time systems, Computer Performance and Dependability Symposium, 1995. Proceedings., International, 24-26 April 1995 .

[12] K.G. Shin, HARTS: a distributed real-time architecture, Computer Volume 24, Issue 5, May 1991.

[13] A. Benso, M. Rebaudengo, M.S. Reorda, P.L. Civera, An integrated HW and SW fault injection environment for real-time systems, Defect and Fault Tolerance in VLSI Systems, 1998. Proceedings., 1998 IEEE International Symposium on 2-4 Nov. 1998.

[14] J. Arlat, Y. Crouzet, J. Karlsson, P. Folkesson, E. Fuchs, G.H. Leber, Comparison of physical and software-implemented fault injection techniques, Computers, IEEE Transactions on Volume 52, Issue 9, Sept. 2003.

[15] M. Prochazka, Software Fault Injector for the DECOS Architecture, diploma thesis, May 2006.

[16] Debian wiki, The Process Address Space, http://wiki.debian.org.tw /index.php/The_Process_Address_Space, 2007.

[17] A. Nayani, M. Gorman, R. S. de Castro, Memory Management in Linux, http://www.ecsl.cs.sunysb.edu/elibrary/linux/mm/mm.pdf, May 2002.

[18] RTAI Official website, RTAI API Documentation, https://www.rtai.org/documentation/magma/html/api/, 2007.

[19] P. Mantegazza, DIAPM RTAI for Linus: WHYs, WHATs and HOWs, Dipartimento di Ingegneria Aerospaziale; Politecnico di Milano, 1999.

[20] E. Bianchi, L. Dozio, P. Mantegazza, A Hard Real Time support for LINUX, Dipartimento di Ingegneria Aerospaziale; Politecnico di Milano, 1999.

[21] M. Masmano, I. Ripoll, A. Crespo, and J. Real, TLSF: a New Dynamic Memory Allocator for Real-Time Systems, Universidad Politecnica de Valencia, Spain., 2004.

[22] P. Puschner, C. Koza, Calculating the Maximum Execution Time of Real-Time Programs, Real-Time Systems. Vol. 1 (2), 1989.

[23] RTAI examples, RTAI: Hard real time test, http://www.captain.at/ programming/rtai/test.php, 2006.

[24] J.C. Laprie, Dependability: Basic Concepts and Terminology in English, Springer-Verlag, 1992.

[25] M. Grottke, K.S. Trivedi, Fighting Bugs: Remove, Retry, Replicate, and Rejuvenate, Duke University, 2007.

[26] K. Trivedi, G. Ciardo, B. Dasarathy, M. Grottke, A. Rindos, B. Varshaw, Achieving and Assuring High Availability, Duke University, 2008.

[27] B. Lindsay, A Conversation with Bruce Lindsay: Designing for failure may be the key to success., Association for Computing Machinery, 2004.

[28] J. Gray, Why do computers stop and what can be done about it, Technical Report 85.7, 1985.

[29] G.A. Kanawati, N.A.Kanawati, J.A. Abraham, FERRARI: a tool for the validation of system dependability properties, Fault-Tolerant Computing, 1992. FTCS-22. Digest of Papers., Twenty-Second International Symposium on 8-10 July 1992.

[30] W.-I. Kao, R.K. Iyer, D. Tang, FINE: A fault injection and monitoring environment for tracing the UNIX system behavior under faults, Software Engineering, IEEE Transactions on Volume 19, Issue 11, Nov. 1993.

[31] R. Schwebe, Echtzeit unter Linux mit RTAI, Elektronik, 2002.

[32] N. Pease, R. Shostak, L. Lamport, Reaching Agreement in the Presence of Faults, Journal of the ACM Vol. 27 (2), 1980.

[33] C. Jones, M.O. Killijian, H. Kopetz, E. Marsden, N. Moffat, M. Paulitsch, D. Powell, B. Randell, A. Romanovsky, R. Stroud, Revised Concepts of DSoS Conceptial Model. Project Deliverable for Dependable Systems of Systems (DSoS), Vienna University of Technology, Real-Time System Group, 2001.

[34] M. Schlager, E. Erkinger, W. Elmenreich, T. Losert, Benefits and Implications of the DECOS Encapsulation Approach, Proceedings of the 8th International IEEE Conference on Intelligent Transportation Systems, 2005.

[35] R. Obermaisser, P. Peti, A Fault Hypothesis for Integrated Architectures, Institute of Computer Engineering, Real-Time Systems Group, Vienna University of Technology, Austria, 2006.

[36] D. Powell, Failure mode assumptions and assumption coverage, Proceedings of the 22nd IEEE Annual International Symposium on Fault-Tolerant Computing (FTCS-22), Boston, USA, 1992.

[37] R.J. Kuntz, Array module connector test program at unisys, MCM '94 Proceedings, San Diego, USA, 1994.

[38] M-C. Hsueh, T-K. Tsai, R-K. Iyer, Fault Injection Techniques and Tools, University of Illinois, 1997.

[39] A. Ademaj, Assessment of Error Detection Mechanisms of the Time-Triggered Architecture Using Fault Injection, Technical University of Vienna, Vienna, 2003.

[40] J. Carreira, H. Madeira, J.G. Silva, Xception: A Technique for the Experimental Evaluation of Dependability in Modern Computers, Dep. Engenharia Informtica, Universidade de Coimbra, Portugal, 1998.

[41] IEC: International Electrotechnical Commission, IEC 61508-1: General Requirments, 1998.

[42] J. Rushby, A Comparison of Bus Architectures for Safety-Critical Embedded Systems, Computer Science Laboratory, Menlo Park CA 94025 USA, 2001.

[43] J. Rushby, Partitioning in the Time-Triggered Architecture, Computer Science Laboratory, SRI International, 2001.

[44] G.J. Whitrow, The Natural Philosophy of Time, Clarendon Press, 1990.

[45] H. Reichenbach, The Philosophy of Space and Time, Dover, New York, 1957.

[46] H. Kopetz, G. Bauer, The Time-Triggered Architecture, Special Issue on Modeling and Design of Embedded Software, 2003.

[47] Verband der Automobilindustrie (VDA), HAWK2015 Herausforderung Automobile Wertschoepfungskette, Henrich Druck + Medien GmbH, 2003.

[48] C.T. Davies, Data processing integrity, In: T. Anderson, B. Randell, Computing System Reliability, Cambridge, University Press, 1979.

[49] P.Peti, R. Obermaisser, F. Tagliabo, A.Marino, S. Cerchio, An Integrated Architecture for Future Car Generations, Proceedings of the Eighth IEEE International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC05), 2005.

[50] H. Kopetz, R. Noussal, Temporal firewalls in large distributed realtime systems, Proceedings of IEEE Workshop on Future Trends in Distributed Computing, Tunis, Tunesia, 1997.

[51] A. Ademaj, H. Kopetz, P. Grillinger, K. Steinhammer, The Time-Triggered Ethernet Protocol Specification, 2005.